ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Repogent: An Autonomous Multi-Agent System for End-To-End Repository Maintenance

Venkata Satya Santhi Somisetti[1], Vijaya Bhaskar Santhuluri[*2], Sai Teja Pathivada[2], Hajiya Jasmine Mohamed[2] and Anitha Kuna[2]

[1]Associate Professor, Department of Computer Science and Engineering (Data Science), Anil Neerukonda Institute of Technology and Sciences, Visakhapatnam, Andhra Pradesh, India

[2]Department of Computer Science and Engineering (Data Science), Anil Neerukonda Institute of Technology and Sciences, Visakhapatnam, Andhra Pradesh, India

DOI: https://doi.org/10.51583/IJLTEMAS.2026.150700076

Received: 29 July 2026; Accepted: 03 August 2026; Published: 13 August 2026

ABSTRACT

Maintaining open-source repositories demands continuous attention to issue triage, code review, build monitoring, and community support—tasks that overwhelm individual maintainers when existing automation tools operate in isolation without shared context. Static analyzers check code quality, CI/CD systems run automated tests, and simple bots handle basic labeling, yet none of these tools share information with each other, leaving maintainers as the sole integration point between disconnected systems. We present Repogent, a multi-agent system where four specialized AI agents—Issue Manager, Pull Request Reviewer, CI/CD Maintainer, and Community Assistant—collaborate through event-driven coordination and persistent cross-task memory. Our GitHub webhook-driven architecture processes events through a priority queue, routes them to specialized agents via intelligent orchestration, and stores decisions in a persistent context layer that enables cross-task reasoning. Events are assigned CRITICAL, HIGH, or NORMAL priority, while a semantic memory module based on sentence-transformer embeddings supports code-level search, and all agents share a common LLM service using Qwen 3 32B accessed through the Groq API. Evaluation on a dataset of 150 issues, 100 pull requests, 80 CI/CD workflow executions, and 120 community queries collected from five active open-source Python repositories demonstrates that Repogent achieves 93.3% accuracy in issue classification, 69.0% combined review coverage for pull requests, 86.7% accuracy in CI/CD failure categorization with a macro F1-score of 0.884, and an average quality score of 3.96 out of 5 for community responses. These results establish a foundation for sustainable open-source maintenance through context-aware, multi-agent workflow automation. Furthermore, the proposed framework demonstrates that multi-agent orchestration with persistent shared memory can serve as an effective and scalable solution for automating repository maintenance activities, improving coordination across heterogeneous development workflows, reducing maintainer overhead, and enabling more consistent decision-making across repository events.

Keywords: Multi-Agent Systems, Repository Automation, Large Language Models, CI/CD Integration, Open-Source Sustainability.

INTRODUCTION

Software maintenance is widely recognized as one of the most resource-intensive phases of the software development lifecycle, often requiring more effort than initial development [5]. Modern open-source repositories receive a continuous stream of bug reports, feature requests, pull requests, build results, and community questions—all competing for the limited attention of project maintainers. This imbalance between incoming workload and available human capacity creates significant bottlenecks in repository management.

Existing automation tools address individual aspects of this problem but operate in isolation. Static analyzers check code quality, CI/CD systems run automated tests, and simple bots handle basic labeling—yet none of these tools share information with each other [10]. As a result, maintainers serve as the sole integration point,

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

manually connecting insights from disconnected tools. This fragmented approach leads to delayed responses, inconsistent decisions, and maintainer burnout.

Recent advances in large language models (LLMs) have opened new possibilities for code understanding at the repository scale. SWE-bench [5] demonstrated that even state-of-the-art models resolve fewer than 2% of real-world GitHub issues when working alone. Multi-agent systems like MAGIS [10] showed that dividing tasks among specialized agents can improve performance significantly—achieving 14% resolution rate through coordinated role assignment. However, these systems remain limited to single task categories and lack persistent memory across different types of repository events.

The integration of AI into software engineering has accelerated rapidly, with tools demonstrating the potential of AI-assisted development. Research on automated program repair [8], LLM-based autonomous agents [12], and multi-agent coordination frameworks such as MetaGPT [3] and AutoGen [13] has further established the viability of intelligent automation for complex software tasks. Pre-trained code models like CodeBERT [2] and benchmarks such as CodeSearchNet [4] have demonstrated effective code understanding capabilities.

Despite this progress, no existing system provides integrated automation across all major repository maintenance tasks while maintaining shared context between them. Current approaches treat each event independently: a build failure is analyzed without knowledge of recent code reviews, and issue classification occurs without awareness of related past discussions. This lack of cross-task context limits the quality and consistency of automated decisions.

Problem Definition

We define the repository maintenance problem as follows. Let $E = \{e1, e2, \ldots, en\}$ denote a continuous stream of repository events, where each event ei belongs to one of four categories $C = \{\text{issue, pull request, workflow run, comment}\}$. Each event carries a payload pi containing metadata such as repository name, author, changed files, and textual content. The objective is to design an automated system S that:

1. Correctly classifies each event ei into its category $ci \in C$,

2. Routes ei to the appropriate processing agent Aj from the set $A = \{Aissue, Apr, Acicd, Aqa\}$,

3. Generates a contextually relevant response ri using both the event payload pi and shared historical context $H = \{(ek, rk) \mid k < i\}$,

4. Updates the shared context store with the new decision for future reference.

The key hypothesis is that maintaining shared context H across all agent interactions improves the quality of individual agent responses compared to stateless independent operation.

Contributions

This paper makes the following contributions:

1. A modular multi-agent architecture for end-to-end repository maintenance, integrating issue triage, code review, CI/CD monitoring, and community support under a single orchestrator.
2. An event-driven coordination framework with priority-based routing and persistent shared memory that enables cross-task contextual reasoning.
3. Specialized agents that combine LLMs with retrieval-augmented generation (RAG) for repository-aware decision making.
4. Evaluation on real-world data from five open-source repositories demonstrating strong performance across issue classification, code review, failure analysis, and community assistance tasks.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Related Work

Jimenez et al., [5] introduced SWE-bench, an evaluation framework designed to test the ability of language models to resolve real-world software engineering problems. The benchmark provides a codebase and a real GitHub issue description, and the model is expected to generate a patch that fixes the issue. SWE-bench was constructed using a three-stage pipeline that collected 2,294 tasks from 12 popular Python repositories. Evaluation is performed by automatically applying the generated patch and running the repository's unit tests, ensuring functional verification. The results revealed that even top-performing models such as Claude 2 struggled, resolving only 1.96% of issues, which demonstrates the complexity of repository-level code editing. The benchmark's limitations include its restriction to Python repositories and its reliance on functional correctness through test passing, which does not measure other important factors such as readability, efficiency, or maintainability. This work establishes the baseline difficulty of repository-level automation and motivates more structured approaches.

Tao et al., [10] proposed MAGIS, a multi-agent framework designed to resolve complex GitHub issues such as those in SWE-bench. MAGIS introduces collaboration between four specialized agents: a Manager to coordinate tasks, a Repository Custodian to locate relevant files, a Developer to implement code changes, and a QA Engineer to review and validate the modifications. The framework operates in two phases. In the planning phase, files are located and a structured plan is created through a simulated kickoff process. In the coding phase, the Developer and QA agents iterate in a loop to produce, review, and verify the code changes. When evaluated on SWE-bench, MAGIS achieved a resolution rate of 13.94%, representing an eight-fold improvement over directly using its base model, GPT-4. However, the framework remains sensitive to prompt design and its evaluation is limited to SWE-bench's Python repositories, so its generalization across programming languages is not fully established. This work demonstrates that role-based decomposition and iterative validation can significantly improve repository-level performance.

Luo et al., [7] proposed RepoAgent, an LLM-powered open-source framework for repository-level code documentation generation. The authors emphasize that developers spend considerable time on program comprehension, making documentation essential for productivity and collaboration. RepoAgent addresses limitations of traditional documentation approaches such as poor summarization, lack of structural guidance, and difficulty maintaining documentation as code changes. The framework employs a three-stage pipeline consisting of global structure analysis, documentation generation, and automatic updates. It leverages AST parsing, the Jedi library, and advanced LLMs such as GPT-3.5, GPT-4, and LLaMA-2, while integrating with Git to synchronize documentation with repository evolution. Experimental results indicate that RepoAgent can generate documentation comparable to or better than human-written content, improving comprehension through structured explanations and examples. Limitations include restriction to Python projects, reliance on LLM accuracy, the need for human oversight, lack of standardized evaluation benchmarks, and concerns around hallucinations, security, and privacy. This work highlights the importance of global repository understanding, which is essential for automation systems that aim to assist users with repository comprehension.

Baqar et al., [1] proposed AI-augmented CI/CD pipelines that embed LLMs and autonomous agents as policy-bounded co-pilots to automate decision-making in software deployment workflows. Their work targets operational bottlenecks such as test triage, security gating, canary analysis, feature flag tuning, and postmortem remediation. The reference architecture includes specialized agents, such as an AI Triage Agent and Observability Agent, orchestrated using agent frameworks like CrewAI and bounded by governance mechanisms such as Open Policy Agent (OPA). The authors propose a four-tier trust model that enables progressive autonomy from providing recommendations to executing full actions. Their evaluation on a React 19 microservice case study uses DORA metrics such as lead time and MTTR, along with AI-specific indicators such as intervention accuracy and human override rate. The study reports improvements including reduced lead time and lower MTTR. Limitations include limited external validity beyond the case study environment, potential measurement bias such as the Hawthorne effect, scarcity of standardized benchmarks for CI/CD agents, and the risk of model drift without continuous retraining. This work supports the motivation for CI/CD-focused automation that can reduce maintainer effort in diagnosing build failures.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Zydroń and Protasiewicz [14] explored automated evaluation of pull request reviews to improve code review efficiency using NLP and machine learning. The study addresses challenges such as manual reviewer assignment and the difficulty of verifying review correctness in large-scale software projects. The approach consists of two main steps. First, an annotation mechanism is developed using pre-trained models such as ChatGPT-4 to extract key review elements from text, including affected code areas, identified issues, and severity. Second, a multi-agent architecture is introduced, where one agent supports review generation and another assesses review quality to ensure consistency and usefulness. The study also discusses existing methods such as RevFinder and TIE and emphasizes combining reviewer expertise metrics with social network signals to improve reviewer assignment. Limitations include dependence on open-source datasets that may not generalize to proprietary settings, challenges in annotation accuracy without human validation, and the need for further empirical evaluation to confirm impact. This work motivates structured and quality-controlled PR feedback systems for practical adoption.

Kallis et al., [6] presented TICKET TAGGER, a GitHub application that uses machine learning to automatically predict and assign labels to new issues. The system relies on the fastText library, selected for computational efficiency to allow deployment on low-end server hardware. The method processes issue titles and bodies as bag-of-words representations and classifies them into common categories such as bug report, enhancement, or question. The tool was validated on approximately 30,000 GitHub issues. A key limitation is the reduced classification performance for “question” issues compared to “bug” and “enhancement,” and another concern is external validity, as the dataset may not represent the full diversity of GitHub repositories. This study demonstrates that automated triage and labeling can reduce maintainer workload, but it also indicates that label ambiguity and repository-specific conventions remain challenges.

Wang et al., [11] proposed RepoMaster, an autonomous agent framework designed to explore and understand GitHub repositories for complex task solving. RepoMaster first performs static hierarchical analysis to construct function-call graphs, module-dependency graphs, and code trees, allowing it to identify core components that are most relevant to a task. It then enters an autonomous exploration and execution loop, dynamically navigating the repository while using information selection techniques to manage the LLM’s limited context window.

The framework was validated on benchmarks such as MLE-R and GitTaskBench. A drawback is the heavy reliance on the static analysis phase, which may be less effective for highly dynamic repositories. Additionally, the system’s generalizability beyond the tested benchmarks remains to be verified. This work supports the importance of structured repository representations and controlled context selection for scalable automation.

Proma and Rosen [9] introduced a prototype system for visual analysis of GitHub issues and commits to provide deeper insights into development patterns. They identified limitations in GitHub’s default interface, which is primarily text-based and makes it difficult to track issue resolution and team contribution patterns. Their tool uses D3.js-based interactive visualizations supported by GitHub API data collection and Python preprocessing. It includes a Timeline View (Gantt chart) to represent issue durations, an Issue Graph (force-directed graph) to link issues with commits and updated files, and summary visualizations that show file update frequencies.

Case studies on repositories such as freeCodeCamp, Hyprland, and Airbnb’s JavaScript style guide suggest that the tool can help identify bottlenecks and bug-prone files. Limitations include lack of support for reopened issues, focus only on file-level updates rather than function-level analysis, reliance on manual or keyword-based linking of commits to issues, and small-scale evaluation with four participants. This work demonstrates that visualization improves situational awareness but does not by itself automate resolution, motivating integrated systems that can transform insights into actions.

Gap Analysis. While each system above addresses a specific aspect of repository management, none provides integrated automation across all four major maintenance tasks with persistent shared context. Table 1 summarizes this comparison.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Table 1: Comparison of related systems with Repogent

System	Issue Triage	PR Review	CI/CD	Community QA	Shared Context
SWE-bench [5]	–	–	–	–	–
MAGIS [10]	✓	–	–	–	Partial
RepoAgent [7]	–	–	–	–	–
Baqar et al., [1]	–	–	✓	–	–
Ticket Tagger [6]	✓	–	–	–	–
RepoMaster [11]	–	–	–	–	–
Repogent	✓	✓	✓	✓	✓

METHODOLOGY

System Architecture

Repogent processes GitHub repository events through three phases:

1. Event ingestion and validation,
2. Intelligent routing and agent execution via a central orchestrator,
3. Repository updates and decision logging through GitHub API integration.

The overall architecture is shown in Figure 1.

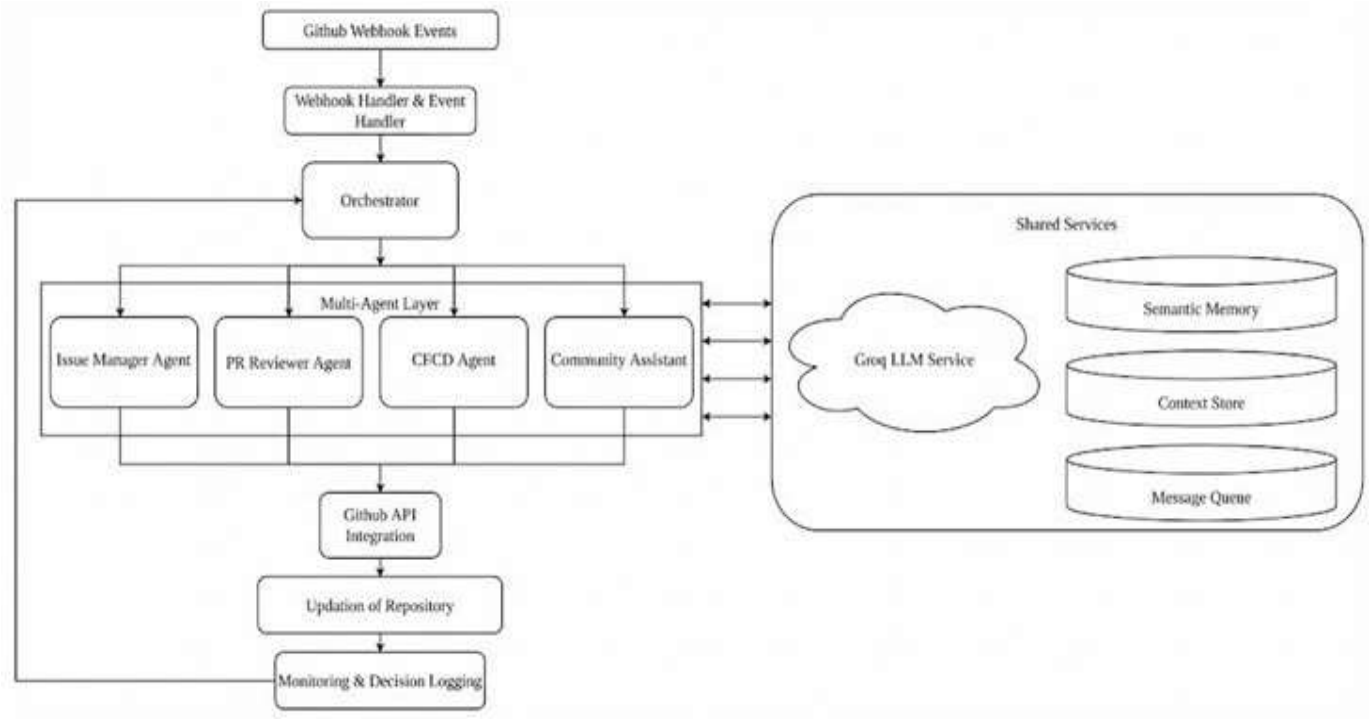

Figure 1: Architecture of proposed model

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Phase 1: Event Ingestion and Validation

The event processing pipeline is triggered whenever a developer performs an action in a GitHub repository, such as opening an issue, creating a pull request, posting a comment, or pushing commits. GitHub generates a webhook payload containing detailed event information, which is delivered to Repogent through GitHub Actions.

The system performs three validation steps:

Event Type Identification. The incoming payload is examined to determine the event category—issue, pull request, workflow run, or comment. This classification determines which agent will process the event.

Metadata Extraction. The full JSON payload is parsed to extract repository name, issue/PR number, contributor identity, modified files, labels, and assignees. All inputs are sanitized to remove invalid or potentially harmful data.

Payload Standardization. The validated information is converted into a standardized internal format containing the event type, repository details, and extracted metadata, then forwarded to the orchestrator. Figure 2 illustrates this stage.

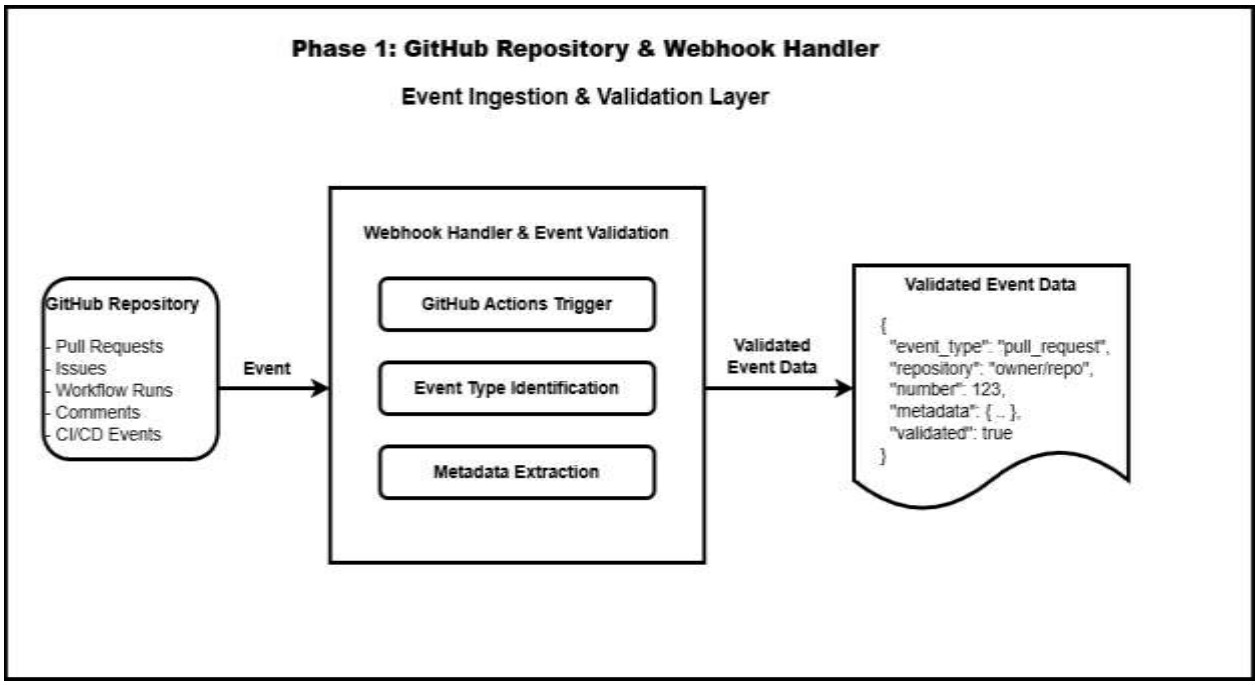

Figure 2: Event ingestion and validation stage

Phase 2: Orchestrator and Multi-Agent Layer

The Orchestrator serves as the central controller, analyzing each validated event and routing it to the appropriate agent:

- Issue Manager Agent — handles new issues (classification, labeling, summarization)
- PR Reviewer Agent — handles pull requests (diff analysis, inline review comments)
- CI/CD Agent — handles failed workflow runs (log parsing, failure categorization)
- Community Assistant Agent — handles comments mentioning @repogent (code search, grounded answers)

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Priority Queue. Events are assigned priority levels: CRITICAL for build failures, HIGH for code analysis tasks, and NORMAL for other events. When the queue reaches capacity, the oldest low-priority messages are removed first.

Context Store. A persistent storage layer maintains records of past reviews, build results, issue classifications, and conversations. Agents query this store before processing new events, enabling decisions informed by repository history.

Shared Services. All agents access common infrastructure including the LLM service (Qwen 3 32B via Groq API), the Context Store, and a Semantic Memory module for code-level search using sentence-transformer embeddings. Figure 3 shows this layer.

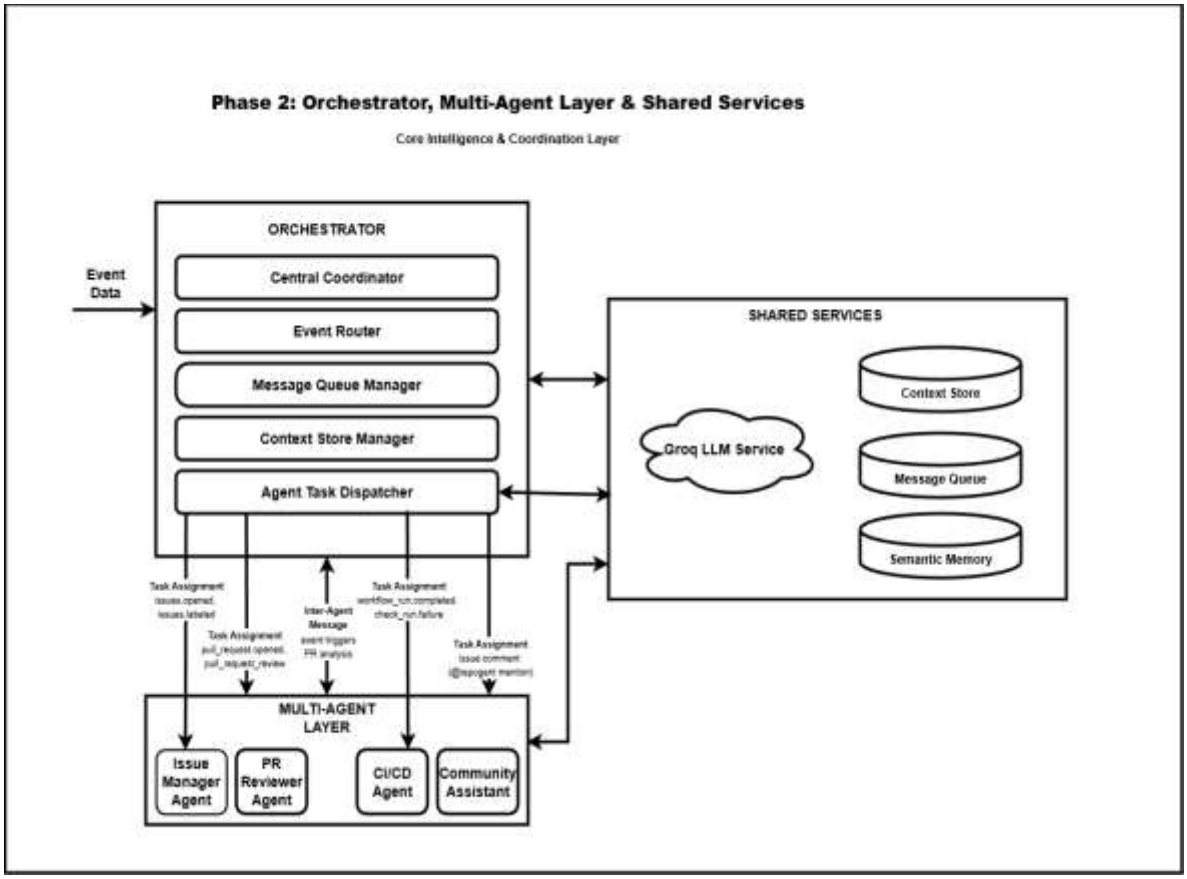

Figure 3: Orchestrator and multi-agent coordination

Phase 3: API Integration and Monitoring

After analysis, each agent sends updates to the GitHub repository through an authenticated API handler:

- The Issue Manager posts classification comments and adds category labels.
- The PR Reviewer creates inline code review comments with severity ratings.
- The CI/CD Agent posts failure analysis with root cause identification.
- The Community Assistant responds with code references and source permalinks.

Every action is recorded in a decision log with timestamps, agent identity, input event, and generated output. This log provides an audit trail and feedback signal for system improvement. Figure 4 shows this layer.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

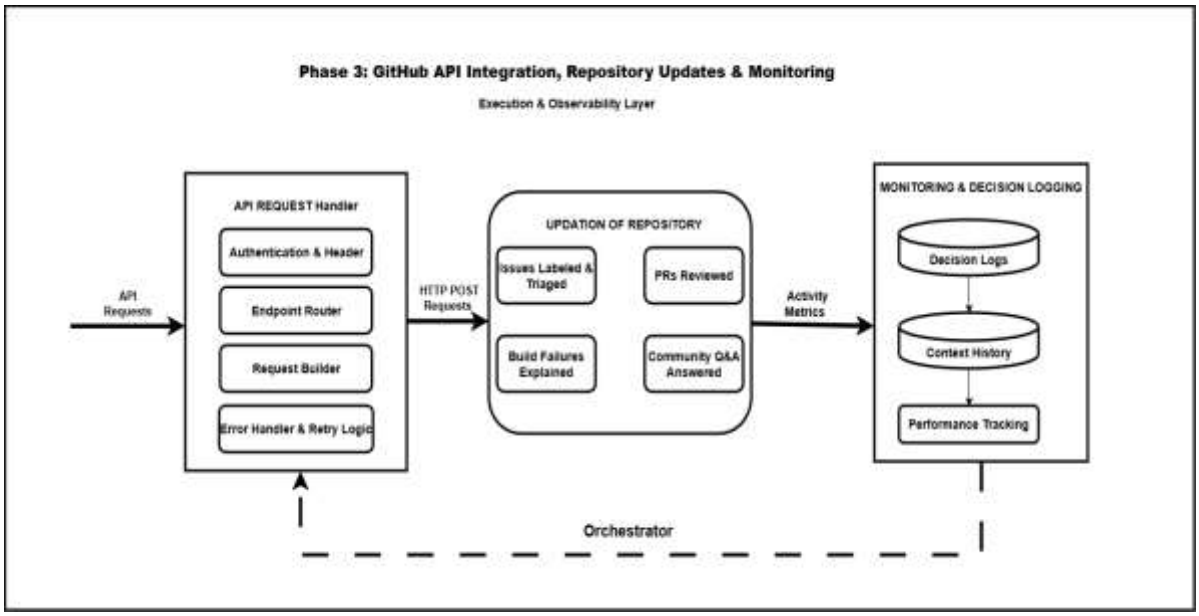

Figure 4: API integration and monitoring layer

Implementation Details

Table 2 summarizes the key technical specifications of the Repogent system.

Table 2: Implementation specifications

Component	**Specification**
LLM Model	Qwen 3 32B (Groq API)
Embedding Model	all-MiniLM-L6-v2
Context Store	JSON-based persistent storage
Queue Capacity	100 messages (priority-sorted)
Webhook Receiver	GitHub Actions (Python scripts)
API Auth	GitHub Personal Access Token
Temperature	0.3 (classification), 0.5 (review)
Max Tokens	1024 per agent response

Algorithmic Workflow

Input: GitHub webhook events (pull requests, issues, comments, workflow runs).

Output: Automated repository actions (labels, reviews, comments, failure analysis).

1. Receive GitHub webhook event via GitHub Actions.
2. Validate event type and extract metadata:
    - Read event type from environment variables.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

- Extract repository name, PR/issue number, author, and changed files.
- Sanitize inputs to prevent injection attacks.

3. Forward validated event to the Orchestrator:
   - Classify event and select target agent.
   - Assign priority level (CRITICAL/HIGH/NORMAL).
4. Enqueue message in priority queue.
5. Agent retrieves message and processes using shared services:
   - Load relevant history from Context Store.
   - Query LLM with structured prompt and context.
   - Generate response (labels, comments, or analysis).
6. Send results to GitHub via authenticated API calls.
7. Log decision with timestamp, agent ID, and response content.
8. Update Context Store for future reference.

EXPERIMENTAL EVALUATION

Dataset and Setup

The evaluation dataset contains multiple public Python repositories. These repositories were selected to represent diverse project sizes, contribution patterns, and development styles. Table 3 summarizes the dataset composition.

Table 3: Evaluation dataset summary

Task	Samples	Source
Issue Classification	150	Opened issues
PR Review	100	Merged PRs with reviews
CI/CD Failure	80	Failed workflow runs
Community QA	120	Curated questions
Total	**450**	

Issues were randomly sampled ensuring balanced representation across Bug, Enhancement, and Question categories. Pull requests were selected from merged PRs that received at least one human review comment. CI/CD failures were collected from workflow runs with non-zero exit codes. Community questions were manually curated to cover common query types including code location, API usage, and architecture questions.

Ground Truth Construction. Original GitHub labels were used as initial ground truth for issue classification. During evaluation, the authors observed that some original labels were inconsistent with issue content (e.g., issues labeled “bug” that described feature requests). To address this, all 150 issues were manually reviewed

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

and corrected where necessary. We report accuracy against both original and corrected labels to ensure transparency.

Each agent is compared against relevant baselines:

- Keyword Heuristics: Rule-based matching using predefined keyword lists for each category.
- TF-IDF + SVM: Traditional text classification using term frequency features, representative of conventional ML approaches.
- Regex Log Parser: Pattern-matching rules for CI/CD log analysis, representing rule-based automation.
- BM25 Retrieval: Sparse keyword retrieval for community QA.
- Static Analysis (Pylint): Automated linting for PR review comparison.

Prompt Design. Each agent uses a structured prompt template with three components: (1) a system instruction defining the agent's role and output format, (2) relevant context retrieved from the Context Store (e.g., past labels for the Issue Manager, recent build logs for the CI/CD Agent), and (3) the current event payload. The Issue Manager prompt specifies three valid labels (Bug, Enhancement, Question) and requires a one-sentence justification. The PR Reviewer prompt includes the diff content and requests line-specific comments with severity levels (critical, warning, info).

Issue Manager Performance

The Issue Manager classifies GitHub issues into Bug, Enhancement, and Question categories using LLM-based semantic reasoning. Against corrected labels, Repogent achieves 93.3% accuracy. The large gap between original (82.0%) and corrected (93.3%) accuracy demonstrates that the LLM correctly identifies maintainer intent even when original labels are noisy—a significant advantage over supervised classifiers that learn from noisy labels. Traditional approaches such as keyword heuristics, which rely on surface-level pattern matching, struggle with ambiguous natural language where the same words appear across different issue types. TF-IDF-based classifiers capture lexical patterns but are similarly affected by label noise in training data.

The precision–recall comparison across methods is shown in Figure 5.

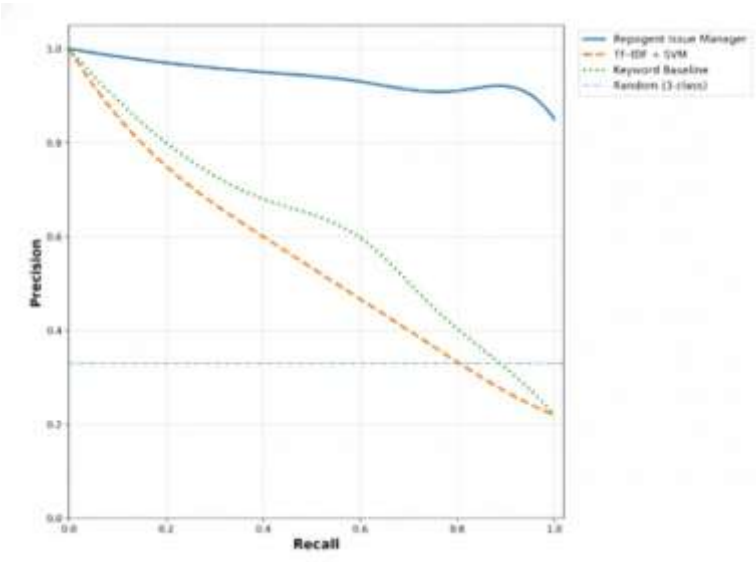

Figure 5: Precision–recall comparison of the Issue Manager and baseline methods

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

PR Reviewer Performance

The PR Reviewer analyzes pull request diffs and generates line-level review comments with severity ratings and fix suggestions. Evaluation uses merged PRs that received human review comments, comparing AI-generated insights against human reviewer behavior.

We define Combined Review Coverage (CRC) as:

$$CRC = |I_{AI} \cap I_{total}| / |I_{total}| \quad (1)$$

where IAI is the set of review insights identified by the automated system and Itotal is the total set of distinct review insights across all sources (human reviewers, AI, and static analysis). A higher CRC indicates that the system captures a greater proportion of meaningful code review observations.

Repogent achieves a CRC of 69.0%, meaning it identifies roughly two-thirds of all meaningful review points. Static analysis tools (Pylint) achieve a CRC of only 12.3%, as they detect formatting and style issues but miss semantic problems such as logic errors, missing edge cases, and architectural concerns.

Notably, manual inspection suggests that the majority of Repogent's review comments raise points not explicitly covered by human reviewers, indicating that Repogent complements rather than duplicates human review effort. The distribution of review insights is shown in Figure 6.

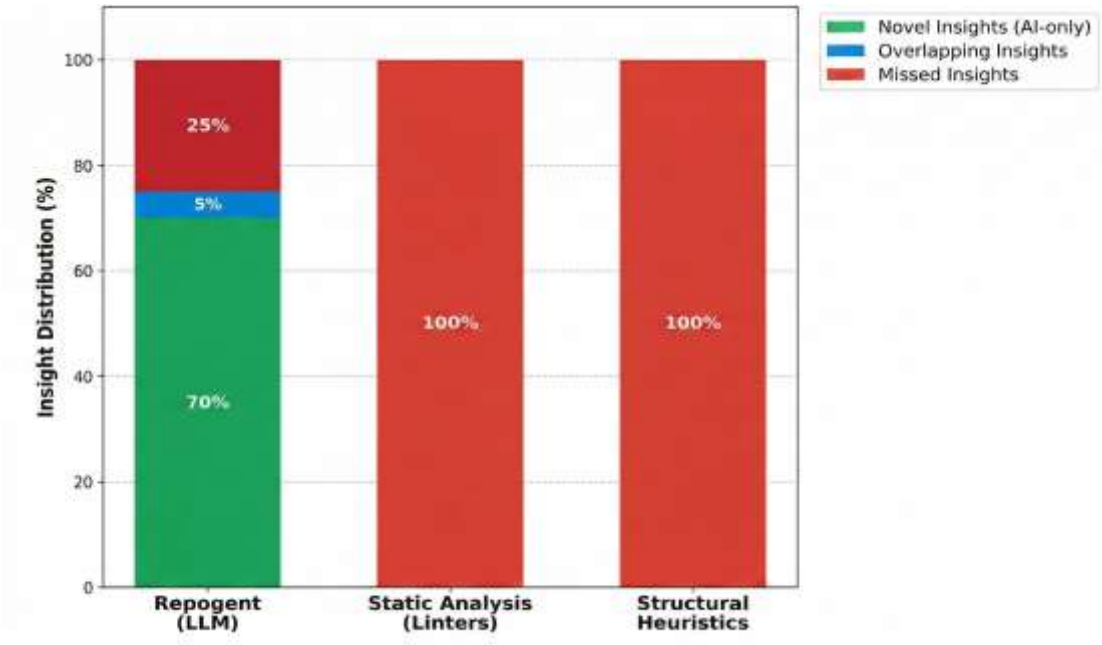

Figure 6: Distribution of review insights across PR review systems

CI/CD Agent Performance

The CI/CD Agent analyzes GitHub Actions logs from failed workflow runs and classifies failures into four categories: test failure, build error, dependency issue, and infrastructure problem.

The agent achieves 86.7% classification accuracy with a macro F1-score of 0.884. The confusion matrix (Figure 7) shows that the agent achieves perfect classification for test failures and dependency issues, while build errors and lint-related failures show some misclassification due to overlapping error patterns in log output. The accuracy comparison across methods is shown in Figure 8.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

The confusion matrix presents results on a representative subset of classified failures across six fine-grained categories.

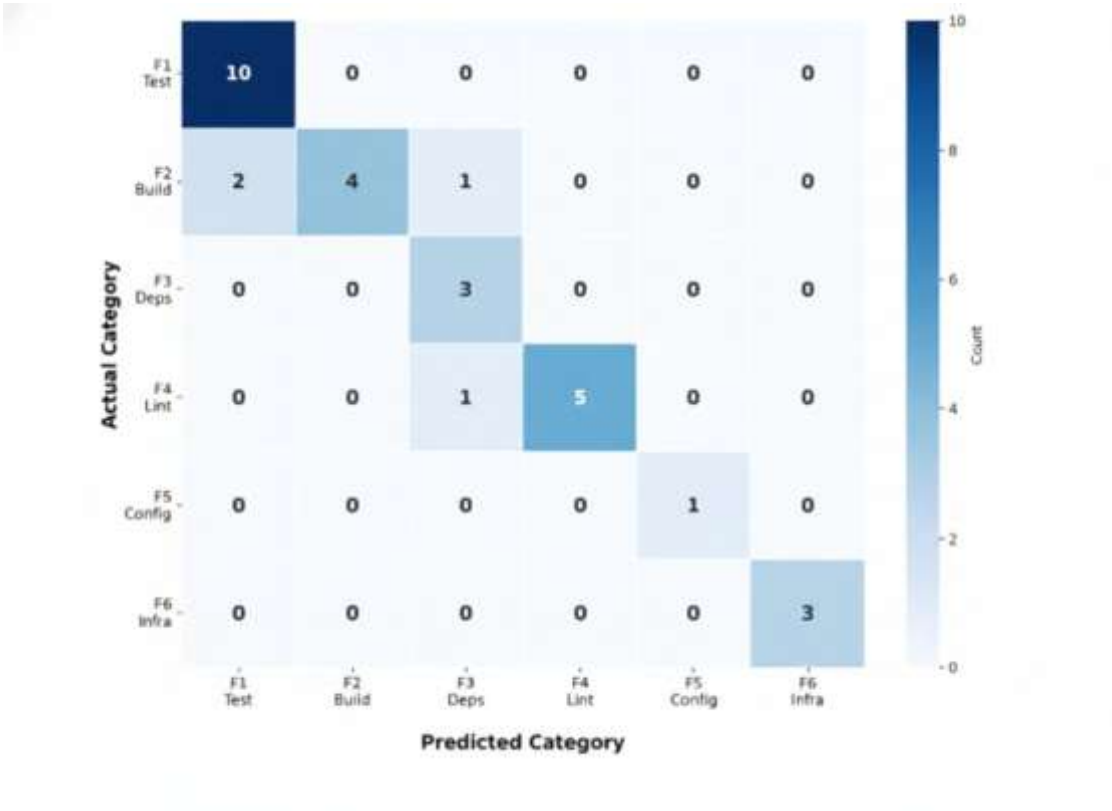

Figure 7: Confusion matrix for CI/CD failure classification

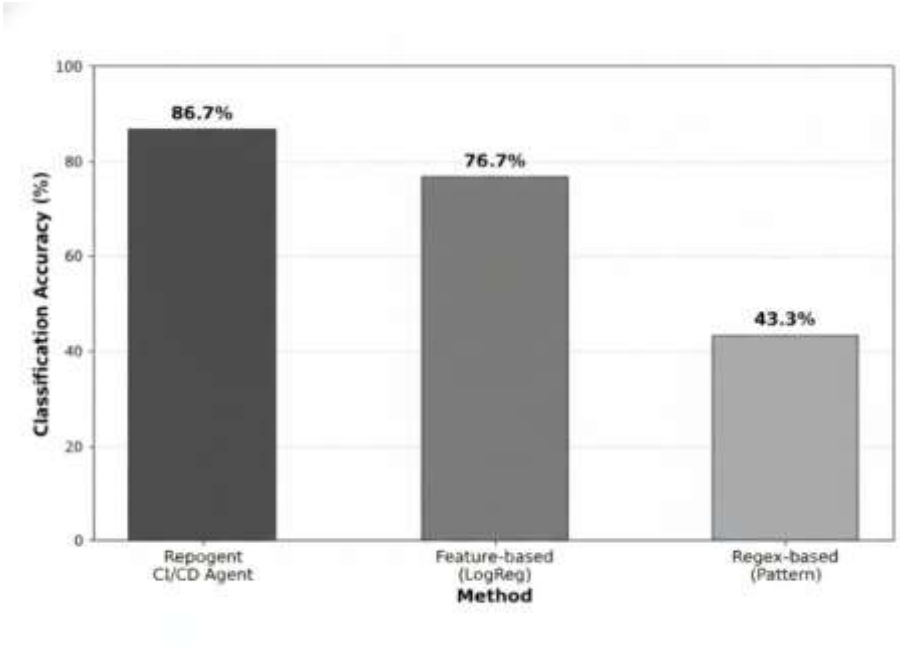

Figure 8: CI/CD failure classification accuracy comparison

Community Assistant Performance

The Community Assistant answers repository-specific questions by retrieving relevant code using semantic search and generating grounded explanations with source permalinks.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Response quality was rated on a 5-point scale by two evaluators (1 = incorrect/irrelevant, 5 = fully correct with complete code references). Inter-rater agreement was $\kappa = 0.82$. Repogent achieves an average score of 3.96/5 with 64% of responses rated as high quality (4), significantly outperforming traditional keyword-based retrieval approaches in response relevance and completeness. The cumulative distribution of scores is shown in Figure 9.

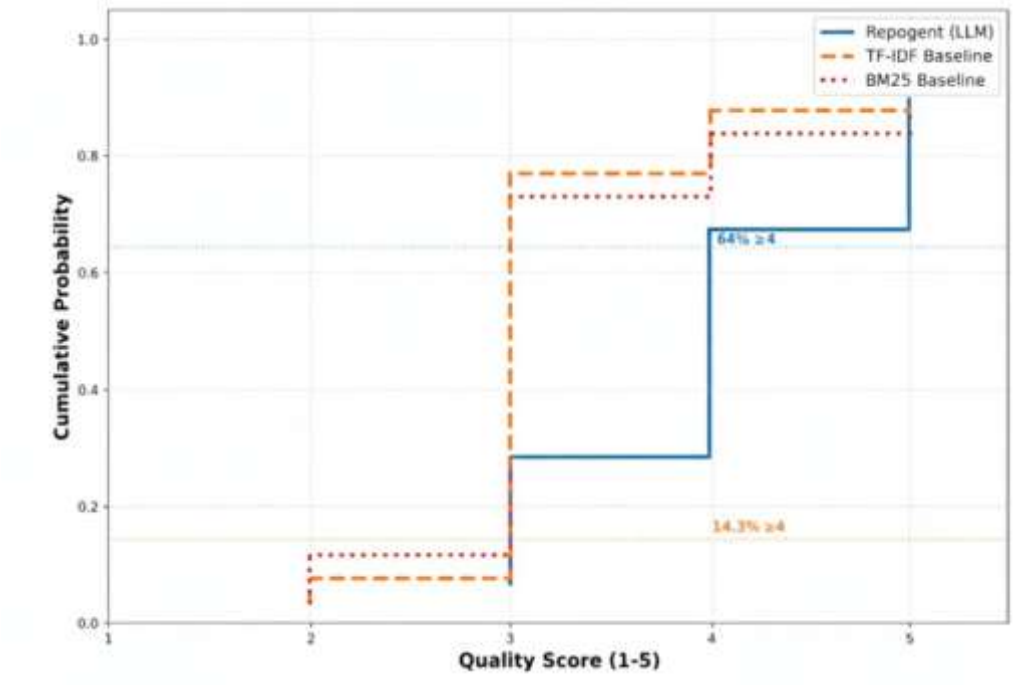

Figure 9: Cumulative distribution of response quality scores for Community Assistant

SYSTEM OUTPUTS

Figures 10–13 present representative outputs from live deployment. Figure 10 shows an incoming issue event captured and routed to the Issue Manager. Figure 11 demonstrates automated labeling and summarization. Figure 12 presents AI-generated PR review comments with severity ratings. Figure 13 shows a community response grounded in repository code with source permalinks.

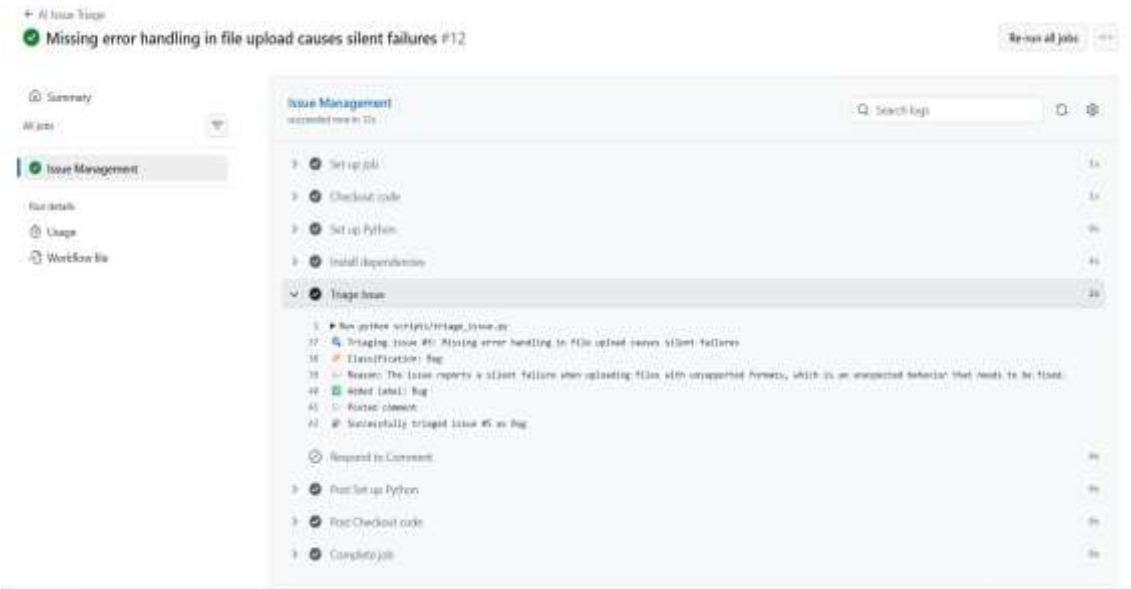

Figure 10: Issue event captured and parsed (Issue Manager triggered)

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

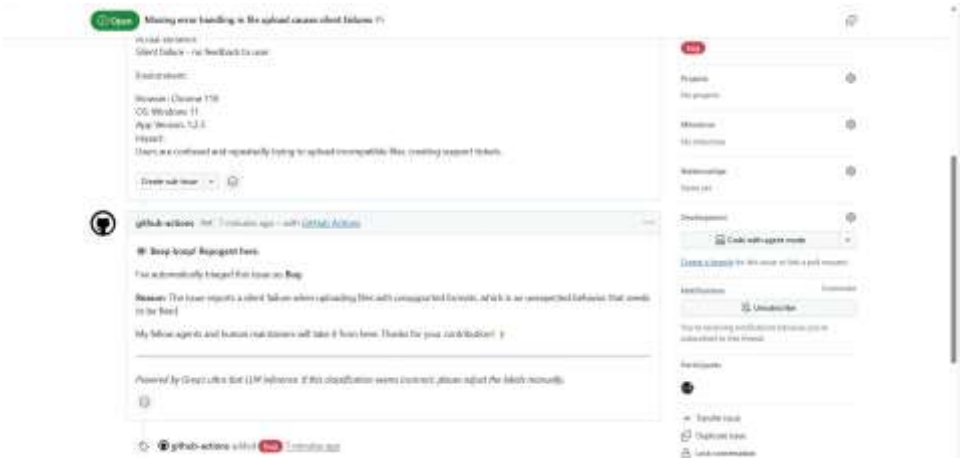

Figure 11: Issue labeled and summarized automatically

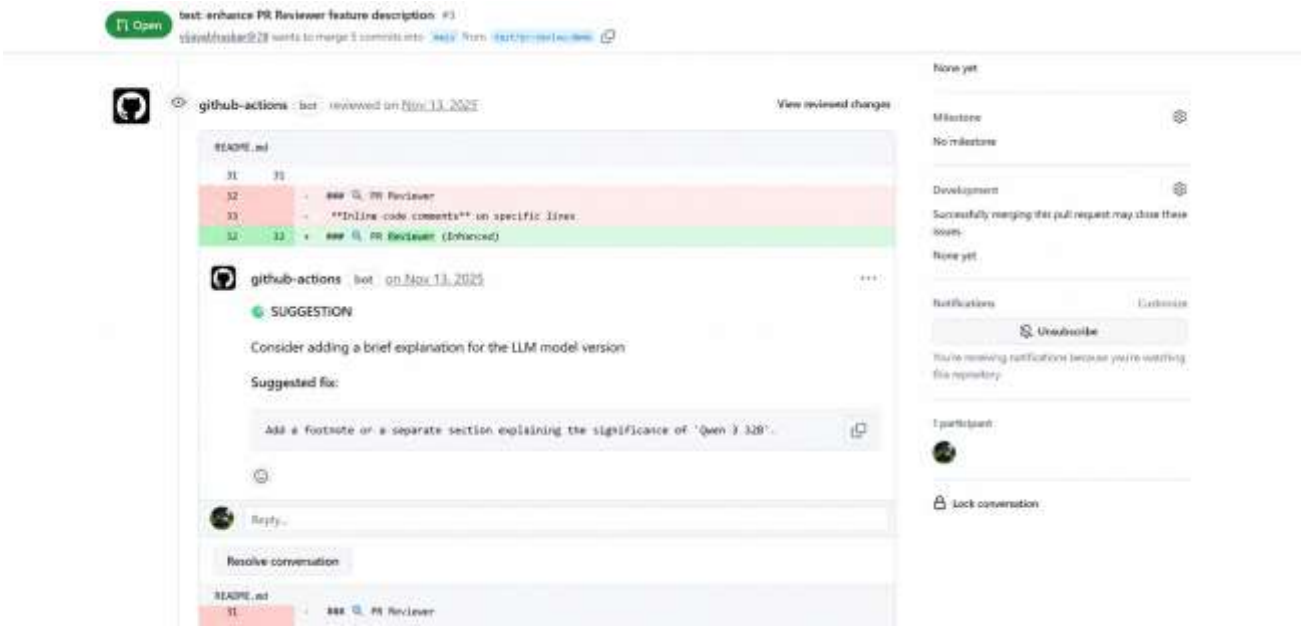

Figure 12: PR review comments generated and posted

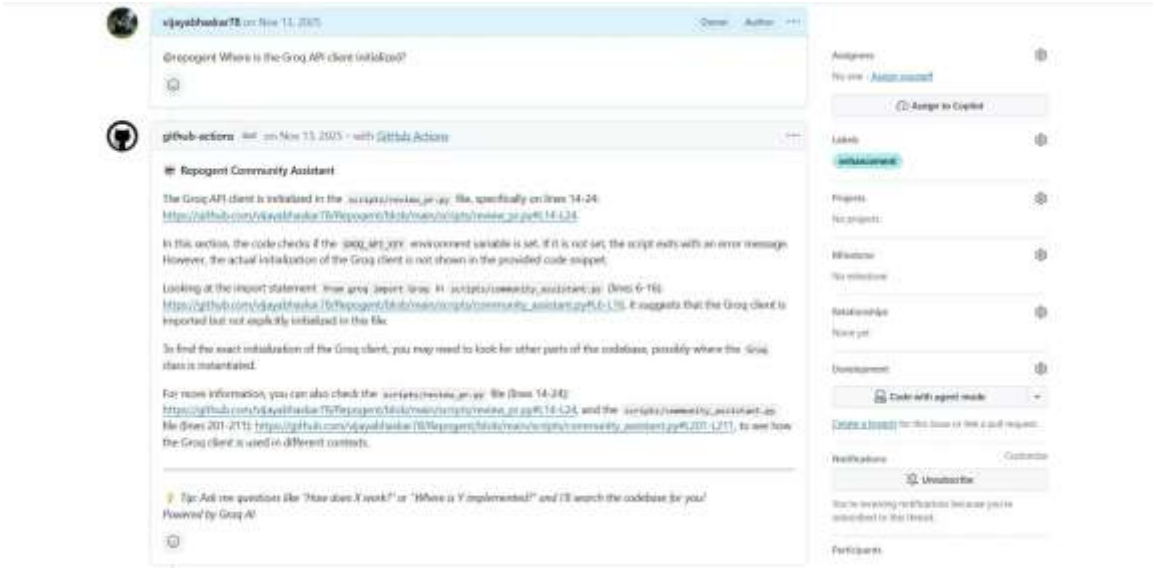

Figure 13: Community assistant repository-grounded response with permalinks

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

DISCUSSION

Key Findings

The experimental results support three main findings. First, LLM-based classification significantly outperforms traditional approaches for issue triage, particularly in the presence of label noise. The gap between original-label accuracy (82.0%) and corrected-label accuracy (93.3%) suggests that general-purpose LLMs with appropriate prompting can identify maintainer intent even when original labels are inconsistent—an advantage over keyword heuristics and statistical classifiers that rely on surface-level pattern matching.

Second, the architectural design of shared contextual memory across agents enables informed decision-making by allowing each agent to access relevant history from other repository activities. While a formal ablation study comparing with-context versus without-context configurations is left for future work, the system design ensures that agents can leverage cross-task information when processing new events.

Third, the PR Reviewer's low overlap with human reviewers combined with high CRC (69%) indicates that AI review is complementary to human review. This suggests that the greatest value comes from deploying AI reviewers alongside—not instead of—human reviewers.

Limitations and Threats to Validity

Single LLM Dependency. Repogent relies on Qwen 3 32B via the Groq API. System availability depends on API uptime, and performance may vary with different models. Future work should evaluate model-agnostic deployment.

LLM Hallucination. LLMs may generate plausible but incorrect suggestions, particularly for complex code review scenarios. All automated actions include clear attribution to indicate AI-generated content.

Dataset Size. The evaluation dataset of 450 total samples, while manually verified, is modest in scale. Results should be validated on larger, more diverse repositories.

Language Scope. The current evaluation focuses exclusively on Python repositories. Generalization to other programming languages requires further investigation.

Cost and Scalability. Each event consumes approximately 1,000–2,000 tokens of LLM input/output. For high-traffic repositories processing hundreds of events daily, API costs and rate limits could become significant constraints.

Ablation Validation. The current evaluation does not include a formal ablation study comparing system performance with and without shared context. While the architecture supports cross-task memory, its quantitative impact on individual agent accuracy has not been isolated experimentally.

Conclusion and Future Work

This paper presented Repogent, a multi-agent system for end-to-end repository maintenance that integrates four specialized agents under a centralized orchestrator with persistent shared memory. The system processes GitHub events in real time, routes them to appropriate agents, and generates automated responses informed by cross-task context.

Experimental evaluation on multiple open-source Python repositories demonstrated strong performance: 93.3% issue classification accuracy, 69.0% combined review coverage for PR review, 86.7% CI/CD failure classification accuracy (macro F1 = 0.884), and an average community response quality of 3.96/5.

Future work will focus on: (1) extending platform support to GitLab and Bitbucket, (2) incorporating policy-based safety mechanisms to prevent harmful automated actions, (3) improving retrieval through vector-indexed

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

semantic search, (4) enabling automated patch generation for common bug patterns, (5) evaluating model-agnostic deployment across different LLMs, (6) validating performance on larger datasets spanning multiple programming languages, and (7) conducting formal ablation studies to quantify the impact of shared contextual memory on individual agent performance.

Abbreviations

AI: Artificial Intelligence; API: Application Programming Interface; AST: Abstract Syntax Tree; BM25: Best Match 25; CI/CD: Continuous Integration / Continuous Deployment; CRC: Combined Review Coverage; JSON: JavaScript Object Notation; LLM: Large Language Model; NLP: Natural Language Processing; PR: Pull Request; RAG: Retrieval-Augmented Generation; SVM: Support Vector Machine; SWE-bench: Software Engineering Benchmark; TF-IDF: Term Frequency–Inverse Document Frequency.

ACKNOWLEDGEMENT

The authors would like to acknowledge the support of Anil Neerukonda Institute of Technology and Sciences, Visakhapatnam, for providing the infrastructure and research environment necessary to conduct this work. The authors also acknowledge the use of AI-based tools for grammar checking and improving the clarity of this manuscript.

Author Contributions

Dr. S.V.S Santhi: Supervision, guidance, proofreading, and approval of the final manuscript.

S. Vijaya Bhaskar: Conceptualization, system design, implementation (majority of coding), experimentation, and manuscript drafting.

P. Sai Teja: Assisted with agent module development, evaluation, and result analysis.

M. Hajiya Jasmine: CI/CD agent design, testing, and data collection.

K. Anitha: Community assistant development, literature review, and manuscript editing.

Every author discussed the results and contributed to the final manuscript.

Conflict of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Declaration of Artificial Intelligence (Ai) Assistance

Generative AI tools were used only for language editing and improving clarity in this manuscript. All technical content, system design, implementation, experimental results, and analysis are the authors' original work. The authors take full responsibility for the accuracy and integrity of the paper.

Ethics Approval

This research did not involve human participants or animal subjects; therefore, ethical approval was not required. All data was collected from publicly available open-source repositories under their respective licenses.

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

REFERENCES

1. M. Baqar, S. Naqvi, and R. Khanda, "AI-augmented CI/CD pipelines: From code commit to production with autonomous decisions," in Proceedings of the 3rd International Conference on Foundation and Large Language Models (FLLM), Vienna, Austria, 2025, doi: https://doi.org/10.1109/FLLM67465.2025.11391007.
2. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547, doi: https://doi.org/10.18653/v1/2020.findings-emnlp.139.
3. S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang et al., "MetaGPT: Meta programming for a multi-agent collaborative framework," arXiv preprint arXiv:2308.00352, 2023, doi: https://doi.org/10.48550/arXiv.2308.00352.
4. H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," arXiv:1909.09436, 2019, doi: https://doi.org/10.48550/arXiv.1909.09436.
5. C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press et al., "SWE-bench: Can language models resolve real-world GitHub issues?" arXiv preprint arXiv:2310.06770, 2023, doi: https://doi.org/10.48550/arXiv.2310.06770.
6. R. Kallis, A. Di Sorbo, G. Canfora, and S. Panichella, "Predicting issue types on GitHub," Science of Computer Programming, vol. 205, art. 102598, 2021, doi: https://doi.org/10.1016/j.scico.2020.102598.
7. Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu et al., "RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation," in Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2024, pp. 436–464, doi: https://doi.org/10.18653/v1/2024.emnlp-demo.46.
8. M. Monperrus, "Automatic software repair: A bibliography," ACM Computing Surveys, vol. 51, no. 1, art. 17, pp. 1–24, 2018, doi: https://doi.org/10.1145/3105906.
9. R. A. Proma and P. Rosen, "Visual analysis of GitHub issues to gain insights," arXiv:2407.20900, 2024, doi: https://doi.org/10.48550/arXiv.2407.20900.
10. W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "MAGIS: LLM-based multi-agent framework for GitHub issue resolution," arXiv preprint arXiv:2403.17927, 2024, doi: https://doi.org/10.48550/arXiv.2403.17927.
11. H. Wang, Z. Ni, S. Zhang, S. Lu, S. Hu, Z. He et al., "RepoMaster: Autonomous exploration and understanding of GitHub repositories for complex task solving," arXiv preprint arXiv:2505.21577, 2025, doi: https://doi.org/10.48550/arXiv.2505.21577.
12. L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang et al., "A survey on large language model based autonomous agents," Frontiers of Computer Science, vol. 18, no. 6, art. 186345, 2024, doi: https://doi.org/10.1007/s11704-024-40231-1.
13. Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv preprint arXiv:2308.08155, 2023, doi: https://doi.org/10.48550/arXiv.2308.08155.
14. P. W. Zydroń and J. Protasiewicz, "Enhancing code review efficiency – Automated pull request evaluation using natural language processing and machine learning," Advances in Science and Technology Research Journal, vol. 17, no. 4, pp. 162–167, 2023, doi: https://doi.org/10.12913/22998624/169576.