Repository-Level Automation for CI Build Repair using Multi-Agent

Rabeya Khatun Muna
Concordia University
Montreal, Canada
rabeykhatunmuna@gmail.com

Sumit Kumar
Concordia University
Montreal, Canada
i.sumitkumar@outlook.com

Abstract

Automated repair of Continuous Integration (CI) build failures remains a critical yet unresolved challenge in modern software development. As the software industry is growing, developing and maintaining large-scale projects is also becoming complex. While CI pipelines are essential for accelerating delivery and ensuring integration correctness, they frequently fail due to issues such as misconfigurations, dependency conflicts, and code inconsistencies. Diagnosing and resolving these failures often requires extensive manual effort, particularly when analyzing unstructured logs and identifying the root cause within large codebases. This paper presents AutoCIRepair, a repository-level multi-agent framework that automates the analysis and repair of CI build failures. The system leverages the reasoning capabilities of Large Language Models (LLMs) by orchestrating three specialized agents: an Error Extractor that converts raw logs into structured error contexts, a Debugger that localizes faults using code diffs and call graph reasoning, and a Developer that synthesizes and validates patches through CI workflow re-execution. The framework is also integrated with the LCA CI Builds Repair benchmark for real-world evaluation, by applying patches in git and triggering the workflow to verify in real-time whether AutoCIRepair successfully produces semantically meaningful and structurally aligned patches for real-world CI failures. By combining symbolic tool usage with LLM-driven analysis in a modular pipeline, AutoCIRepair advances the field of automated program repair in CI environments, offering a scalable solution to enhance reliability and reduce developer intervention.

CCS Concepts

• **Software and its engineering** → **Automatic programming**; **Maintaining software**; **Software testing and debugging**; • **Computing methodologies** → **Natural language processing**.

Keywords

CI build repair, Fault Localization, Automated Program Repair, Multi-agent system, Large Language Models, Log Analysis, Git

SOEN 7481'25, Montreal, QC, CA

ACM Reference Format:
Rabeya Khatun Muna and Sumit Kumar. 2025. Repository-Level Automation for CI Build Repair using Multi-Agent. In .

1 Introduction

Continuous Integration (CI) has become the backbone and one of the fundamental practices in modern software engineering by supporting the rapid and collaborative development of software systems. It allows developers to integrate code changes frequently into a shared repository, enables early detection of integration issues, improves software quality, and accelerates release cycles [21]. Each commit triggers an automated pipeline that may contain project compilation, configuration setup, execution of unit and integration tests, and code quality checks. The generation of detailed build logs captures the outcome of each stage [4]. These logs are critical for diagnosing build failures and provide developers with valuable insight into the behavior of the system during integration.

As CI adoption continues to grow, particularly in large-scale and fast-paced development environments, the frequency and complexity of integration tasks have increased significantly. It has also been reported that the Google monolithic repository reportedly handles over 5,500 commits daily, resulting in the execution of more than 100 million test cases [8]. In such environments, a single faulty commit can lead to a series of downstream failures if not addressed early stage. Consequently, these failures can increase system instability and introduce undetected bugs, making the overall codebase more difficult to manage. Moreover, these disruptions delay the validation of subsequent changes. As a result, developers have to spend a significant amount of time analyzing, identifying, and resolving bugs within large and complex codebases. These slow down the entire development cycle, reduce productivity, and accumulate unresolved build failures. Ultimately, it leads to long-term technical debt, reducing maintainability, and software quality.

Despite the advantages offered by CI systems, the task of locating and correcting the root causes of build failures remains largely manual. Traditional fault localization techniques often depend on structured test results or detailed bug reports to identify problematic areas in the source code. However, CI build failures frequently arise from a wide variety of issues—including configuration errors, test flakiness, or environment mismatches—and are often captured only in unstructured or semi-structured log data [1, 8]. Developers need to read lengthy log outputs of failed CI builds thoroughly, identify relevant error messages, analyze the project to identify bug locations or faulty code, and write patches to fix the errors. This entire process is time-consuming, cognitively demanding, and becomes increasingly challenging as projects scale in size and complexity.

Recent advances in machine learning and natural language processing have introduced new opportunities for automating software maintenance tasks. Large Language Models (LLMs), trained on vast corpora of source code and technical documentation, have demonstrated the ability to understand programming syntax, interpret compiler and runtime errors, and even suggest context-aware code edits [6, 14]. These capabilities have led to early successes in automated bug fixing and code generation. However, existing approaches primarily focus on local code snippets or test-based repair scenarios and rarely address the unique challenges posed by CI environments—particularly the interpretation of raw build logs and the mapping of errors to faults at the repository level.

To address these limitations, we propose a novel multi-agent framework that leverages the reasoning power of LLMs to automate CI failure analysis and program repair. The system is powered by GPT-4o-mini and coordinates multiple specialized agents, each responsible for a discrete task in the CI debugging workflow. These include analyzing error logs, identifying suspicious files, localizing faulty code segments, and generating minimal patches. The framework is designed to work directly with unstructured CI build logs, extract structured insights, and synthesize program repairs in a fully automated manner. By automating the interpretation of CI logs and the generation of fixes, this approach aims to reduce the manual effort required for debugging, improve the responsiveness of development teams to integration failures, and ultimately enhance the reliability and maintainability of software projects.

The significant contributions of this research are summarized as follows:

- We introduce a novel Large Language Model (LLM)-driven multi-agent framework powered by GPT-4o-mini, designed to automate the end-to-end process of fault localization and program repair by analyzing unstructured CI build failure logs.
- We aim to build a multi-agent system that autonomously retrieves error context, identifies fault locations, and generates candidate patches to reduce debugging time from unstructured CI logs, improve CI/CD pipeline efficiency, and enhance software maintainability.
- We have done an extensive evaluation of the effectiveness of the automated multi-agent based framework by integrating with the LCA CI Build Repair Benchmark. The evaluation involves applying the generated patches using git apply and re-running the original GitHub Actions workflows to determine whether the CI failures are resolved.
- We conduct a user study to compare the generated patches with ground-truth developer fixes.

2 Background

2.1 Continuous Integration (CI) and Build Failures

Continuous Integration (CI) automates the process of building, testing, and deploying code after every commit, aiming to reduce integration risk and accelerate feedback. However, modern CI systems are highly complex: they orchestrate multi-stage pipelines using tools like GitHub Actions, Jenkins, or GitLab CI, which involve scripts, environment setup, and third-party dependencies. This complexity often leads to build failures unrelated to code logic, including YAML misconfigurations, toolchain incompatibilities, caching issues, and flaky job runners.

Recent studies report that more than 30% of builds in active projects fail at least monthly [12, 28]. These failures generate multi-file logs spanning hundreds of lines, often with limited semantic structure, making manual diagnosis slow and error-prone. Traditional debugging tools are designed for static or single-file faults and cannot interpret the distributed signals embedded in CI logs, diffs, and configurations. Thus, there is a growing need for systems that can parse, reason over, and act upon raw CI artifacts.

2.2 Fault Localization and Program Repair

Program repair typically begins with fault localization—the task of identifying buggy program elements responsible for observed failures. Spectrum-Based Fault Localization (SBFL) uses statistical correlations between test outcomes and execution coverage to compute suspiciousness scores [11]. While SBFL is computationally cheap and easy to deploy, it assumes high-quality test coverage and test oracles—assumptions often violated in CI environments, where failures may occur during setup, configuration, or before tests are run.

To mitigate these limitations, learning-based techniques such as DeepFL, CombineFL, and Grace apply machine learning models or GNNs to incorporate additional signals (e.g., code metrics, historical faults, call dependencies) [15, 26, 27]. However, they require large labeled datasets and still rely on test-level instrumentation, which is often unavailable in CI pipelines. Moreover, they are not equipped to reason over mixed-format inputs like YAML files or logs.

2.3 LLMs and Agent-Based Structured Reasoning

Recent advances in Large Language Models (LLMs) have shown strong performance on software engineering tasks. Pretrained on source code, documentation, and bug reports, LLMs like GPT-4 and Claude 3 can perform zero-shot bug explanation, log summarization, and code synthesis [5, 23]. Their ability to integrate natural and programming languages makes them promising tools for CI debugging, where logs often mix shell commands, error messages, and filenames.

However, vanilla prompting suffers from limitations: long logs can exceed context windows, fragile prompts can lead to hallucinations, and LLMs lack persistent memory and tool interfaces. To address these challenges, LLM agents have emerged—systems that wrap LLMs in structured control loops, assign them roles, and enable tool use.

Frameworks like Toolformer [25] and LangChain [13] allow LLMs to invoke symbolic tools (e.g., file readers, AST extractors, diff viewers), shifting low-level parsing away from fragile prompts. Recent agent-based systems like LLM4FL [19] and DeepAgent [2] show that assigning specialized roles (e.g., retriever, analyzer, patcher) and enabling inter-agent reasoning leads to more scalable and modular pipelines, especially in large repositories. These systems use feedback loops (e.g., verbal reinforcement learning) to refine outputs across phases.

Such multi-agent LLM frameworks are particularly well-suited to CI repair, where failure diagnosis requires navigating across logs, diffs, and code—and the fix must be tested by re-running the pipeline.

3 Related Work

3.1 Overview and Motivation

Research on CI repair spans several axes: classical fault localization (e.g., SBFL), learning-enhanced models (e.g., DeepFL), LLM-based reasoning (e.g., LLMAO, SoapFL), and modular multi-agent systems (e.g., LLM4FL, DeepAgent). However, most systems do not target CI-specific artifacts such as YAML, raw logs, and GitHub Actions pipelines—and even fewer produce executable patches.

Table 1 compares prior systems across five key capabilities: CI awareness, agent modularity, patch generation, log handling, and tool use. Most systems lack CI log integration and do not complete the repair loop.

3.2 Spectrum-Based and Learning-Based Localization

SBFL remains lightweight but assumes test-level feedback and deterministic failures. FLUCCS [9], DeepFL [15], and CombineFL [27] improve upon SBFL by incorporating metrics and learning-based ranking. Grace [26] and DepGraph [16] model interprocedural relations using GNNs. However, all of these methods are designed for static, test-case-driven environments—not dynamic CI systems with logs, YAML errors, or misconfigured job runners.

3.3 LLM-Based Fault Localization and Prompting

LLMAO [29] and ChatRepair [23] prompt LLMs with logs or stack traces for fault prediction, but suffer from token length limits and lack end-to-end automation. LLMLocate [10] and SoapFL [22] apply retrieval-augmented prompting to improve scale but remain monolithic and non-modular.

LLM4FL [19] introduces multi-agent fault localization with graph-based navigation and verbal reinforcement learning, but only targets test failures in Java and lacks CI context awareness.

3.4 CI Log Analysis and Failure Classification

LogChunks [17] contributes a labeled dataset of CI log segments, while Catching Smells [12] catalogs common YAML anti-patterns in GitHub Actions. UniLoc [18] applies IR and AST segmentation for CI localization but stops short of synthesis. CI-Evolution [7] shows how minor changes in toolchains and configuration can frequently break CI.

3.5 Multi-Agent and Tool-Augmented Architectures

Multi-agent repair systems like DeepAgent [2] and Agentic RAG [24] structure LLM reasoning into modular roles and planning graphs. Toolformer [25] and LangChain [13] allow LLMs to invoke tools, improving robustness. These frameworks demonstrate that combining symbolic tools with LLMs enables better reasoning and scalability. However, most are designed for general code understanding or synthetic benchmarks—not real-world CI debugging.

Table 1 presents a comparative overview of AutoCIRepair and existing CI repair approaches. The analysis demonstrates that our proposed framework—AutoCIRepair—distinctly targets CI build failures through a coordinated pipeline of three specialized agents: the Error Extractor, the Debugger, and the Developer. Each agent is augmented with symbolic tool access, including AST parsing, Git diff analysis, and YAML configuration checks. This architecture enables the system to process raw CI logs, analyze indirect call relationships, generate diff-compatible patches, and automatically rerun CI pipelines to validate the applied fixes. AutoCIRepair is among the first frameworks to fully integrate fault localization with CI-based validation, offering a closed-loop solution suited for real-world CI workflows. The comparison in Table 1 underscores its unique capabilities and practical relevance across diverse CI failure scenarios.

4 Methodology

In this section, we introduce a multi-agent-based automated system for analyzing CI build failure logs, fault localization, and generating patches to automatically repair bugs using the repository-level dataset lca-ci-builds-repair.

Figure 1 provides an overview of our technique. Our proposed approach uses several tools and prompting techniques to complete the entire workflow from analyzing logs to fixing bugs by using three LLM-Agents: (i) Error Context Extraction Agent, (ii) Debugger Agent, and (iii) Developer Agent, to iteratively analyze errors from logs, fault localization, and generate patches for fixing bugs. The introduced multi-agent-based automated system operates over a repository-level benchmark dataset LCA-CI-Builds-Repair, which consists of failed GitHub Actions runs, with each datapoint including the repository information, a failed commit SHA, workflow metadata, and raw log output. The first agent, Error Context Extraction Agent uses the context retrieval strategy in chunking and a memorization way to retrieve the error details from large logs. After that, the Debugger Agent uses the error details and the diff of the failed commit to generate bug reports containing fault localization. In the end, the Developer agent uses the bug report and generates patches with validation. Below, we provide more details on each of these phases.

4.1 Error Context Extraction Agent

This agent is designed to extract, structure, and interpret meaningful error information from raw, unstructured system logs. It follows a chunk-based division and multi-phase analysis process that enables precise error localization, categorization, and workflow mapping to assist in effective debugging and system reliability assessment.

4.1.1 Chunk-Based Log Division Phase: To process large and complex logs efficiently, the Error Context Extraction Agent first performs a chunking phase. The entire raw log is divided into manageable segments based on token count, log structure, or time-based separation to ensure each chunk remains within the processing constraints of the LLM's input context. In this stage, a given log

Aspect	AutoFL	LLM4FL	UniLoc	LogChunks	DeepAgent	AutoCIRepair (Ours)
Input Type	Test results, coverage	Test logs, call graph	CI logs, source files	CI logs	Logs, code	Logs, diffs, CI configs
CI-Aware	No	No	Yes	Yes	Partial	Yes
Uses AST / Graphs	No	Yes (call graph)	Yes (AST split)	No	Yes (GNN/DAG)	Yes (call graph expansion)
Agent-Based Modularity	No	Yes (multi-agent)	No	No	Yes (tool agents)	Yes (3-role agent system)
Handles Logs & YAML	No	No	Partial	Yes	Yes	Yes
Produces Patches	No	No	No	No	Yes (static patches)	Yes (CI-tested patches)
Handles Token Limitation	N/A	Yes (grouped inputs)	Yes (chunked AST)	Yes (log chunking)	Yes (tool-driven)	Yes (per-agent scoped input)
Open-Ended Reasoning	No	Limited	No	No	Yes	Yes
Repair Validation	N/A	N/A	N/A	N/A	No	Yes (CI rerun)

Table 1: Comparative summary of CI fault localization and repair systems. AutoCIRepair uniquely integrates CI awareness, agent modularity, AST reasoning, and end-to-end patch validation.

with millions of lines is divided into subsets $L_1, L_2, \ldots, L_k$, such that the size of each subset satisfies:

$$|L_i| \leq \text{Token Limitation}$$

This step guarantees that the model can process each log chunk entirely within its attention window. The segmentation strategy also preserves the local temporal ordering of log entries, which is critical for capturing causality and co-occurrence patterns around error events.

4.1.2 Error Identification and Contextual Extraction Phase: Each chunk is passed through the agent to identify and extract error lines along with their surrounding context. These include log entries immediately preceding and following the error message, which often contain stack traces, warning signs, or system state information. The agent analyzes those error lines, retrieved from logs by using pattern-based and semantic filters to identify explicit error markers (e.g., Exception, Error, Traceback), log severity levels(e.g., FATAL, WARN), and keyword proximity and co-occurrence relevant to error propagation and provide structured error details containing commit, error context from raw logs, step name, where the workflow has failed and retired error files, mentioned in the log.

4.1.3 Workflow-Aware Error Analysis Phase: Following context extraction, the agent performs workflow-aware analysis by aligning the identified errors against a predefined execution or operational workflow. This allows for a deeper understanding of where and why failures occur. The agent analyzes the structure error context with workflow. This analysis leads to structured error details that contain error summary with natural language, Classifying errors into types (e.g., configuration error, Assertion error, Linting error), and File association for identifying the specific file(s) likely responsible for the error based on stack trace parsing, jobs detail relevant to error and provide details of the part that needs to focus on. These well-structured error details are used for further steps for downstream debugging, fault localization, and automated recovery mechanisms.

4.2 Debugger Agent

The Debugger Agent is a fault localization module responsible for analyzing error context and code changes to identify the exact lines of code responsible for CI build failures. It leverages multiple tools in a structured multi-phase flow that includes error log analysis from the previous agent and file change information of failed commits from GitHub. These informations are used as suggestions to be suspicious files, code content retrieval, and iterative fault localization, with optional call graph expansion. This agent ensures a precise, scalable, and autonomous localization process for complex CI failures across large codebases.

4.2.1 GitHub-Based File Change Integration: The Debugger Agent uses file changes as input via a Git-based module. This module extracts all file changes between the failed commit (`sha`) and its immediate predecessor (`sha_fail~1`):

$$\text{Diff}_{\text{commit}} = \texttt{git diff}(\text{sha}_{\text{fail}}\texttt{~1}, \text{sha}_{\text{fail}})$$

Each file is parsed using a unified diff extractor and stored in structured JSON format. These files and their before and after code are used during the analysis of the suspicious files suggestion phase to ensure that any modifications lead to the failure of the commit.

4.2.2 Error-Aware File Retrieval Phase: At the initial stage, the agent consumes structured error outputs from the *Error Context Extraction Agent*. It uses this information to identify related files mentioned in logs, which are the primary location of being faulty. Each file is searched within the local repository, fetched the entire content of that file is fetched for further analysis.

$$E = \{e_1, e_2, \ldots, e_n\}$$

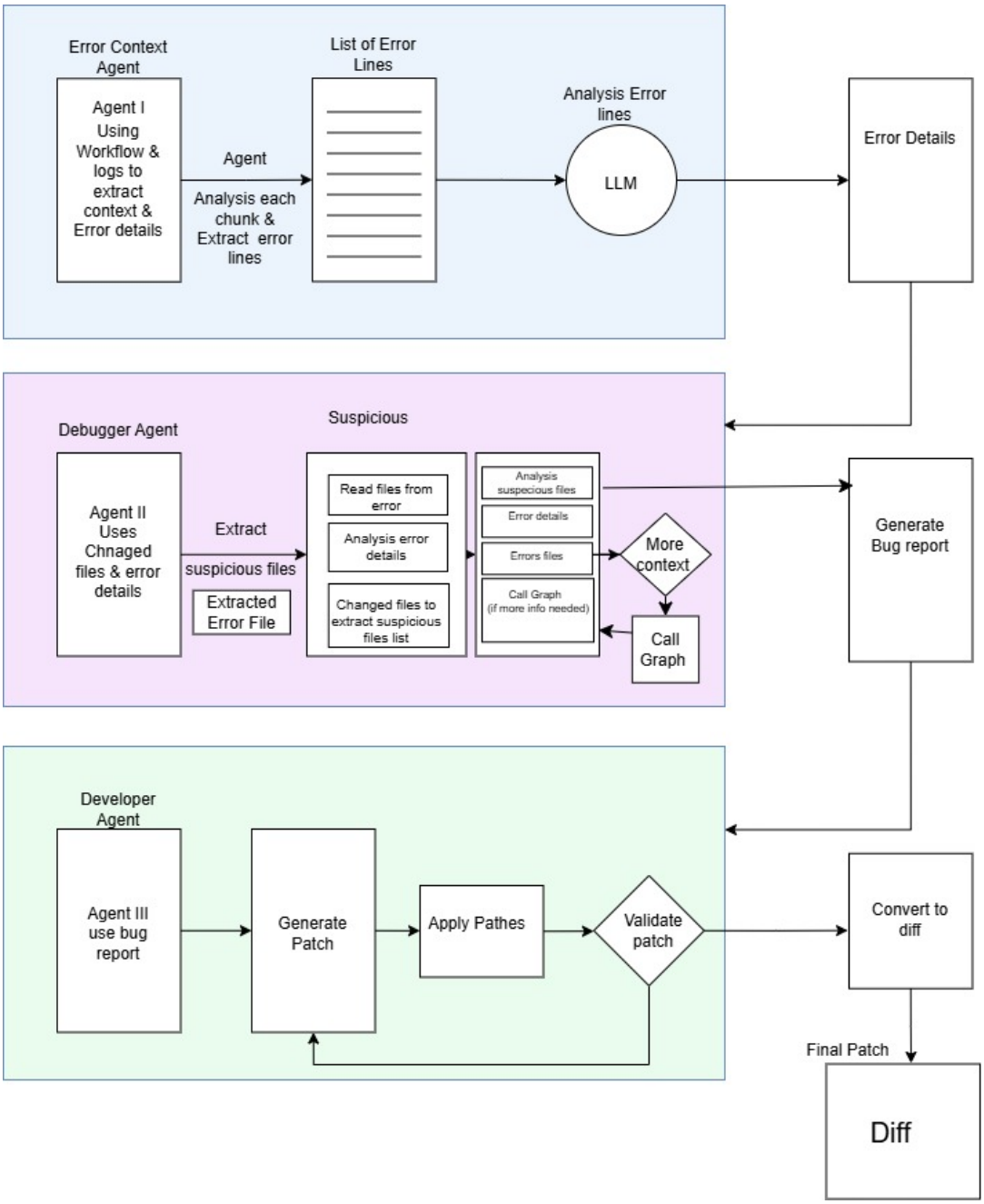

Figure 1: An overview of AutoCIRepair: An Automated Multi-Agent system for CI Build Repair.

E is denoted as the list of structured error entries. For each

$$e_i \in E,$$

The agent extracts file paths.

$$F_i = \{f_{i1}, f_{i2}, \dots\}$$

To retrieve source content, the git checkout applies to ensure these files are available and read-accessible in the checked-out commit environment.

4.2.3 Suspicious File Suggestion and Content Expansion Phase: In this phase, Agent calls a tool, that uses both the structured error summaries and diff-based file change information retrieved from the failed commit (via git diff) to analyze which files are most suspicious.

It considers the semantic content of code changes (after_code), error file content if it is mentioned in error details, and their relation to error messages for analyzing to identify which files are most likely to be faulty and select list of suspicious files $S = \{s_1, s_2, \dots\}$ where each s_i includes: File path, Full File Content, Changed Code from Diff and Reason for suspicion. This phase integrates data from both error_logs and GitHub-based changed_files_content using chunk-based processing concerning the LLM token window.

4.2.4 Context Expansion via Call Graph (Conditional Phase): The phase is conditional; If the fault cannot be confidently localized due to insufficient context while analyzing the suspicious files and content against error details, the agent triggers *call graph expansion*. This is requested explicitly when the model returns `"requires_call_graph_expansion": true`. The agent provides relevant methods or class names from the suspicious files to use for the *Call Graph Extractor* to generate caller-callee relationships and merges this extended dependency context with existing analysis and re-runs fault localization.

Here, $C = \{c_1, c_2, \dots\}$ is denoted as the list of targeted classes or methods name, and let G_i be the subgraph for c_i:

$$\text{Final Context} = \text{Error Context} + \text{Suspicious Files} + \bigcup_{i=1}^{|C|} G_i$$

This graph-based information helps for higher precision in identifying indirectly faulty components and complex inter-file issues. The call graph is constructed using static analysis based on Python's Abstract Syntax Tree (AST), which enables the extraction of structural code relationships such as function definitions and invocations. These caller-callee relationships form the backbone of the dependency graph, allowing the system to trace the flow of execution and understand how different components interact. This additional context is especially useful when dealing with subtle bugs that span across multiple functions or files.

4.2.5 Fault Localization Phase: The agent combines the extracted error context, error file content, suspicious file content, and caller-callee relationship content from the graph(if required) into a unified prompt. This content is chunked if necessary and passed to the LLM for precise fault localization. The model identifies the minimal faulty line range $[l_{\text{start}}, l_{\text{end}}]$, associated code snippet, and reason for fault in each location. The output is returned in a strict JSON format, including:

- `file_path`
- `fault_localization_level` (e.g., file-level, method-level)
- `location_type` (e.g., import, class, config)
- `line_range` and `code_snippet`
- `suggested_fix`

This bug report, which contains detailed information on fault localization, guides the *Developer Agent* in automated patch generation.

Overall, the agent performs fault localization in an automated and structured manner using a Langchain-based agent. It sequentially invokes a predefined set of tools—`read_error_file`, `suggest_and_read_suspicious_files`, `final_fault_localiz-ation`, and conditionally `expand_context_with_callgraph`—to systematically analyze error contexts, retrieve code changes, and localize faults by caching intermediate results to ensure both efficiency and consistency throughout the localization process.

4.3 Developer Agent

The Developer Agent is responsible for generating, validating, and formatting patches that address the faults identified during prior analysis. Operating in an automated and modular pipeline, the agent synthesizes fixes from structured bug reports, verifies their correctness through CI workflows, and converts validated fixes into a standardized diff format for integration. The entire process is orchestrated using a LangChain-based agent framework with minimal human intervention.

4.3.1 Patch Generation Phase: At the initial stage the Developer Agent uses the bug report, curated from previous agents, to convert the structured bug reports into concrete code edits. Upon initialization, the agent retrieves bug location details, including file paths, line ranges, faulty code snippets, and natural language explanations of the underlying issue. The agent groups suggestions by file and constructs a prompt that incorporates: Faulty code segments, Root cause summary, High-level fix intent, and Full file context. The agent then synthesizes a fully modified version of the file containing only the minimal necessary changes to resolve the described issue. The response is strictly formatted as a complete file after applying patches, wrapped in triple backticks to enable parsing. Files with no effective changes are skipped, and all modified files are preserved for the validation phase.

4.3.2 Patch Validation Phase: After generating candidate patches, the Developer Agent attempts to validate them by running the original repository's CI workflow locally using GitHub Actions via the act tool. This ensures that the patch not only fixes the identified issue but also preserves the correctness of the software. During validation, the modified files are written to disk, and then the CI workflow file is executed via subprocess to verify whether the workflow passes or not after applying the patches. If the workflow is successful, the patch is accepted, otherwise, a new patch is attempted after restoring the files as a feedback loop.

4.3.3 Patch Formatting Phase: Once a patch has passed validation, the agent proceeds to convert the changes into a unified diff format. For each modified file, the original and patched lines are compared using standard diff algorithms. The resulting diff is structured in Git-compatible format, including $a/file$ and $b/file$ headers, and makes it compatible for Git apply for evaluation patches in the lca-ci-build-repair benchmark. This ensures that the generated patches are directly usable by downstream CI repair benchmarks and patch application tools.

5 Evaluation

In this section, we first present the studied dataset, benchmark, and evaluation metrics. Then, we present the answers to the research questions.

Studied Dataset Benchmark: To evaluate the effectiveness of our CI build repair system, we utilize the benchmark provided by the Long Code Arena (LCA), a suite of real-world CI failure scenarios curated by JetBrains Research [3]. Specifically, we use the

lca-ci-builds-repair dataset [20], which comprises 68 genuine CI build failure cases drawn from 32 open-source Python repositories hosted on GitHub. Each data point includes a failing commit along with its corresponding fix commit, forming realistic failure–repair pairs that emulate authentic developer workflows. The dataset captures not only the CI logs from the failed runs but also Git diffs, repository metadata, and file-level modifications. The CI logs are segmented by workflow step and typically include actionable error messages, system outputs, or tracebacks—allowing detailed failure analysis.

The failures span a range of CI-specific issues, including YAML misconfigurations, missing package dependencies, incorrect import statements, and violations of code quality or formatting rules. The dataset is designed to challenge models across diverse repair scenarios while maintaining fidelity to real-world development pipelines. Notably, this benchmark facilitates an end-to-end evaluation by encouraging patch synthesis and automated CI reruns for validation, distinguishing it from earlier fault localization datasets that lacked realistic execution feedback.

Evaluation Metrics: Our primary evaluation metric is *Pass@1*, defined as the proportion of CI failures successfully repaired by the first generated patch. A patch is considered successful if it applies without conflict to the original failing commit and leads to a fully passing CI pipeline upon re-execution. This mirrors developer expectations where the initial fix attempt is often deployed for validation. Complementing this, we also report *Patch Validity Rate*—the percentage of generated patches that are syntactically correct and compile error-free—and *Log Filtering Precision*, which measures the system's ability to extract concise, relevant log segments while discarding extraneous noise. These metrics jointly evaluate the accuracy, reliability, and practical usability of the system in continuous integration workflows.

Implementation and Environment: Our system is implemented in Python 3.13, with all agent orchestration and prompt executions handled via the LangChain framework. The multi-agent architecture consists of three cooperating modules—Error Extractor, Debugger, and Developer—each powered by OpenAI's GPT-4o-mini model (April 2024 release). These agents operate in a structured pipeline and exchange intermediate reasoning artifacts, including filtered logs, AST representations, and patch suggestions. Git operations and diff management are handled through GitPython, while the Debugger Agent integrates AST parsing for lightweight static analysis. To empirically validate repairs, each patch is applied to the failing commit, committed to a forked version of the original repository, and tested via GitHub Actions CI reruns.

All inference runs are deterministic (temperature = 0) to ensure reproducibility. Token usage is minimized through prompt compression and per-file chunking. This controlled environment ensures consistent evaluation while preserving the modularity and scalability of the system.

5.1 RQ1: How does the Multi-agent system-generated patches compare to human-written code?

Motivation. In this research question, we aim to evaluate the similarity between patches generated by our multi-agent system with the written patches by human developers. The goal is to assess how well the system replicates developer intent and structure when repairing CI build failures. While correctness and validation are essential, structural alignment with real-world developer changes offers deeper insight into model quality. This comparison helps identify whether agents are mimicking developer reasoning or simply generating syntactically valid fixes.

Approach. We use the official lca-ci-builds-repair dataset, which contains ground-truth patches under the diff column for each failed commit. For each task, our Developer Agent generates a candidate patch based on the bug report and fault localization provided by upstream agents. We then compare the model-generated patch against the human-written patch using diff similarity analysis. Specifically, we calculate textual similarity, file coverage, and operation types (e.g., reorder import, fix method signature, config changes). Table **??** shows an example where both agent-generated and human patches restructured imports and fixed incorrect paths in Python code.

Result. Out of 68 attempted repairs, 28 patches were successfully generated by the multi-agent system in the first attempt (Pass@1). We evaluated 9 of these patches by comparing them to the corresponding ground-truth developer patches using structural similarity metrics. The comparison focused on line-level similarity using 'difflib' and token-level similarity using Jaccard index over tokenized diffs. While no exact matches were observed—likely due to stylistic and formatting differences. Out of 28, only 8 patches are demonstrated for moderate structural alignment. These included categories such as import reordering, path corrections, and formatting adjustments, which are often not addressed by traditional test-based APR systems. The results are summarized in Table 2 and Table 3.

	Total issues	Generated patches	Total failed
Pass@1	68	28	40

Table 2: Summary of multi-agent system patch generation.

TASK ID	Exact Match	Line Similarity	Token Similarity
4	x	0.834	0.864
5	x	0.725	0.738
98	x	0.454	0.517
142	x	0.188	0.311
143	x	0.041	0.231
23	x	0.019	0.079
89	x	0.023	0.095
169	x	0.129	0.188
Summary	**0/27 (0.0%)**	**0.086**	**0.160**

Table 3: Similarity analysis of 28 multi-agent patches vs. human-written diffs to demonstrate alignment with ground truth.

6 Future Plan

Although our multi-agent system shows a unique approach in automatically repairing CI failures, there are several directions yet to be explored for strengthening and evaluating the performance of automated CI build repair. From analyzing unstructured logs to generating patches, the entire process needs to be efficient and scalable enough to make it applicable and scalable in the real world. For this, we aim to expand the evaluation to understand how effectively the system can perform fault localization and bug repair across more complex CI failures. We also plan to perform a fine-grained analysis to validate how effectively our proposed framework can perform fault localization to identify failure cases and generate patches. We want to validate the performance of agents in each phase, collaborating across different failure types. Moreover, we will scale the dataset to include a more diverse range of CI errors, such as YAML configuration issues, linting violations, dependency conflicts, and cross-language CI tasks. Our goal is to test the system's generalization ability beyond Python-based projects, to improve its robustness in CI-specific error scenarios, and to reduce the computational overhead by optimizing prompt engineering and improving agent interaction efficiency. This includes minimizing token usage, caching intermediate results, and parallelizing subtasks across agents. These improvements will help us create a faster, more scalable repair pipeline suitable for real-world adoption.

7 Conclusion

In this paper, we introduced AutoCIRepair, a novel LLM-driven, multi-agent system for automating CI build repair through structured fault localization and patch generation. Unlike prior approaches that operate on isolated code snippets or rely solely on test coverage, AutoCIRepair addresses the challenges of real-world CI debugging by integrating log analysis, GitHub diff of failed commits, call graph reasoning, and workflow re-execution into a unified, agent-based pipeline. Each agent operates autonomously yet collaboratively to extract error context, localize faults with precision, and synthesize validated patches in a Git-compatible format. We have done an empirical evaluation using the LCA CI Builds Repair benchmark illustrates the feasibility of this approach. We have also demonstrated the generated patches with the growth truth of diff for analyzing alignment with developer-authored fixes with the proposed framework-generated diff. Beyond successfully generating and validating repairs, AutoCIRepair demonstrates structural alignment between its synthesized patches and the ground truth developer fixes, suggesting that the system can replicate developer intent and reasoning to a significant degree. This alignment highlights the framework's potential for integration into real-world CI workflows with minimal human intervention. While the current system shows require further enhancement for its performance across more complex, multi-file, or cross-language CI failures remains an open challenge. Future work will focus on scaling the system to larger benchmarks, enhancing agent collaboration and thinking ability to generate patches more profoundly, and reducing computational overhead through prompt optimization and parallelized tool usage. Ultimately, this research moves toward the vision of intelligent, self-healing CI pipelines capable of autonomously diagnosing and repairing software defects at scale.

References

[1] Henri Aïdasso, Mohammed Sayagh, and Francis Bordeleau. 2025. Build Optimization: A Systematic Literature Review. *arXiv preprint arXiv:2501.11940* (January 2025). https://arxiv.org/abs/2501.11940

[2] Arindam Basu, Vaibhav Krishnan, et al. 2025. DeepAgent: A modular, hierarchical LLM agent framework for long-horizon planning. arXiv:2502.00350 https://arxiv.org/abs/2502.00350

[3] Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, and Timofey Bryksin. 2024. Long Code Arena: a Set of Benchmarks for Long-Context Code Models. *arXiv preprint arXiv:2406.11612* (2024).

[4] Carolin E. Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. 2020. LogChunks: A Data Set for Build Log Analysis. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. ACM, 583–587. doi:10.1145/3379597.3387485

[5] Fan Chen, Xinyu Liu, Chenggang Xu, et al. 2023. Repairing code errors via retrieval-augmented in-context learning. arXiv:2305.13534 https://arxiv.org/abs/2305.13534

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, et al. 2021. Evaluating Large Language Models Trained on Code. OpenAI Technical Report. https://openai.com/research/code

[7] Cong Gao and Shane McIntosh. 2020. CI-evolution: Investigating the evolution of continuous integration workflows in the wild. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. IEEE, 376–386.

[8] Foyzul Hassan, Na Meng, and Xiaoyin Wang. 2023. UniLoc: Unified Fault Localization of Continuous Integration Failures. *ACM Transactions on Software Engineering and Methodology* 32, 6 (November 2023), 1–31. doi:10.1145/3593799

[9] Songyu He, Xiang Chen, et al. 2020. FLUCCS: Using code and change metrics to improve fault localization. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 1391–1393.

[10] Anshul Jain, Dipanjan Roy, et al. 2024. LLMLocate: Accurate fault localization in large codebases with retrieval-augmented LLMs. arXiv:2310.04498 https://arxiv.org/abs/2310.04498

[11] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. ACM, 467–477.

[12] Mohammad Khatami, Gustavo Pinto, Andrea Iannone, and Christoph Treude. 2023. Catching smells in the act: A GitHub Actions workflow investigation. In *Proceedings of the 2023 ACM Joint ESEC/FSE*. ACM, 101–113.

[13] LangChain Contributors. 2023. LangChain: Framework for developing applications powered by LLMs. Retrieved from https://www.langchain.com/.

[14] Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. 2023. Explaining Competitive-Level Programming Solutions using LLMs. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NL-RSE)*. https://arxiv.org/abs/2307.05337

[15] Jiajun Li, Ziyuan Wang, Xue Han, Yiling Lou, and Xinyu Wang. 2021. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE, 654–666.

[16] Xin Liu, Yifan Zhang, Hailong Sun, et al. 2023. DepGraph: A dependency graph-based approach for fault localization. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. ACM, 182–193.

[17] Sagnik Mehta, Pranjal Jain, and Baishakhi Ray. 2024. LogChunks: A dataset for fine-grained log analysis and CI failure reasoning. arXiv:2403.15852 https://arxiv.org/abs/2403.15852

[18] Ankit Mukherjee, Akash Lahoti, Kevin J. Bowers, and Bimal Viswanath. 2023. UniLoc: Unified fault localization in CI/CD pipelines using IR and static analysis. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. ACM, 2041–2052.

[19] Ali Rafi, Jiawei Xu, Saeed Ahmad, and William G. J. Halfond. 2024. LLM4FL: A multi-agent approach to fault localization via graph-based retrieval and reflection. arXiv:2409.13642 https://arxiv.org/abs/2409.13642

[20] JetBrains Research. 2024. LCA CI Builds Repair. https://huggingface.co/datasets/JetBrains-Research/lca-ci-builds-repair. Accessed: April 2025.

[21] Vincent Uchenna Ugwueze and Joseph Nnaemeka Chukwunweike. 2025. Continuous Integration and Deployment Strategies for Streamlined DevOps in Software Engineering and Application Delivery. *International Journal of Computer Applications Technology and Research* 14, 01 (January 2025), 01–24. doi:10.7753/IJCATR1401.1001

[22] Yifan Wang, Xiaohui Wang, et al. 2023. SoapFL: Signature-guided prompting for LLM-based fault localization. arXiv:2306.01534 https://arxiv.org/abs/2306.01534

[23] Zhen Wu, Sheng Yu, Xinyi Zhang, et al. 2022. ChatRepair: Fixing code with LLMs via prompting. arXiv:2209.14545 https://arxiv.org/abs/2209.14545

[24] Yutong Xiong, Qi Liang, and Yichong Zhu. 2025. Agentic Retrieval-Augmented Generation: A survey of multi-agent RAG systems. arXiv:2502.04644 https://arxiv.org/abs/2502.04644
[25] Diyi Yao, Yujie Hou, John Turian, et al. 2023. Toolformer: Language models can teach themselves to use tools. arXiv:2302.04761 https://arxiv.org/abs/2302.04761
[26] Zhiqiang Yu, Yang Xie, Zhenchang Xing, et al. 2022. Grace: Graph-based context representation for fault localization. In *Proceedings of the 30th ACM Joint ESEC/FSE*. ACM, 1240–1251.
[27] Cheng Zhang, Yun Liu, et al. 2022. CombineFL: Fault localization via ensemble learning on program spectra. arXiv:2207.12345 https://arxiv.org/abs/2207.12345
[28] Ting Zhang, Ali Mesbah, and Karthik Pattabiraman. 2018. Systematic evaluation of failures in continuous integration. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 359–369.
[29] Tongtong Zhang, Mingyu Xie, et al. 2023. LLMAO: Token-efficient LLMs for localizing code bugs. arXiv:2307.10184 https://arxiv.org/abs/2307.10184