Journal of Technology Informatics and Engineering (JTIE)
Vol. 4 No. 1 April 2025
E-ISSN: 2961-9068; P-ISSN: 2961-8215, Pages 190-214
DOI: https://doi.org/10.51903/jtie.v4i1.484

LLM-Driven CI Failure Diagnosis and Automated Repair: From GitHub Actions Logs to Patch Recommendation

Hanqi Zhang*[1]
Email: hz0102@yahoo.com
[1]*Computer Science, University of Michigan at Ann Arbor, MI, USA*
*Corresponding Author

Abstract
Continuous Integration (CI) pipelines surface regressions early but also produce long, noisy logs. Diagnosing a failing GitHub Actions run and drafting a safe repair patch can be time-consuming, especially when dealing with dependency drift or configuration errors. We study a practical CI-repair pipeline decomposed into three measurable tasks: (1) coarse failure-type classification, (2) retrieval-based repair (log similarity reuse the closest historical fix diff), and (3) constrained patch generation that emits a unified diff via template+slot filling. The pipeline follows the schema and task framing of JetBrains-Research's lca-ci-builds-repair dataset from Long Code Arena (212 samples). Because runtime restrictions in our environment prevent downloading the original Hugging Face-hosted parquet files, all quantitative results in this paper are evaluated on a locally generated proxy dataset, CI-Repair-Sim212, which matches the benchmark's field schema and evaluation protocol. On CI-Repair-Sim212, failure-type classification reaches a ceiling (Macro-F1=1.000), whereas repair-pattern prediction remains harder (Macro-F1=0.796 with log+workflow). For patch recommendation, retrieval achieves Token-F1@1=0.898 and Pattern@1=0.783 when combining logs with workflow context, and constrained generation further improves diff similarity to Token-F1=0.923. Across tasks, adding workflow YAML context yields consistent gains, motivating hybrid CI assistants that prioritize retrieval when near-duplicate failures exist and fall back to constrained generation when close matches are absent.

Keywords: *CI/CD, GitHub Actions, Failure Diagnosis, Automated Program Repair, Retrieval-Augmented Generation.*

I. INTRODUCTION

Continuous Integration and Continuous Deployment (CI/CD) have become the default operational backbone of modern software engineering. CI pipelines automatically build, test, lint, and package changes to prevent regressions from entering mainline branches (Fowler, 2006; Hilton et al., 2016). However, CI systems also introduce a new operational workload: each failure event demands triage, localization, and repair. In large repositories with many workflow steps, failures range from genuine programming bugs to fragile infrastructure issues such as dependency drift, transient network errors, insufficient permissions, or changes in hosted runners. Even when fixes are straightforward (e.g., pinning a dependency version or adding a missing system package), the time-to-repair can be long because developers must interpret verbose logs, locate the relevant workflow configuration, and craft a patch.

Large language models (LLMs) and agentic tooling have recently demonstrated strong capabilities for code understanding and code editing, including the generation of diffs that apply minimal changes to satisfy tests or static checks (Brown et al., 2020; OpenAI, 2023; Silalahi et al., 2022). A natural next step is to operationalize these models in CI/CD. Once a workflow run

Received on January, 2025; Revised on February, 2025; Accepted on February, 2025; Published on April, 2025

Journal of Technology Informatics and Engineering (JTIE)
Vol. 4 No. 1 April 2025
E-ISSN: 2961-9068; P-ISSN: 2961-8215, Pages 190-214
Journal Link: https://jtie.stekom.ac.id/index.php/jtie/index

fails, the system should automatically (i) pinpoint the failing step, (ii) classify the failure into a repair-relevant category (dependency installation, test failure, lint failure, permission errors, build toolchain issues, etc.), (iii) consult a memory of past failures to retrieve similar incidents and their known fixes, and (iv) produce a candidate patch diff that can be reviewed and merged. This vision is actively explored by industrial and academic work on AI assistants for debugging, software maintenance, and automated program repair (APR) (Weimer et al., 2009; Long & Rinard, 2016).

Despite growing interest, CI failure diagnosis differs from traditional APR in at least three ways. First, CI failures frequently arise from configuration and environment mismatches rather than code semantics. Second, the relevant evidence is distributed across artifacts: the run log, the workflow YAML, dependency manifests, and test configurations. Third, the repair objective is operational: the primary goal is to make the CI workflow pass under the repository's intended constraints, which may require modifying workflows, build scripts, or version pins rather than program logic. For this reason, benchmark datasets that pair CI failure logs with ground-truth fixes are critical for measuring progress.

This paper investigates a reproducible CI-repair pipeline motivated by the JetBrains-Research lca-ci-builds-repair dataset, a benchmark released as part of Long Code Arena that focuses on tasks requiring long-context reasoning (Bogomolov et al., 2024). The dataset contains 212 samples; each sample includes a failed GitHub Actions workflow log, the workflow file content, and a successful fix diff. These annotations enable both failure-type classification and supervised diff generation. While the dataset itself is hosted on Hugging Face and can be loaded with the Datasets library, the execution environment used for this work blocks outbound dataset downloads at runtime. Therefore, we execute full experiments on a local proxy dataset that matches the original schema and evaluation protocol, and we provide an implementation blueprint that can be applied to the original dataset when data access is available. Unless otherwise stated, all quantitative results reported in this paper are computed on CI-Repair-Sim212 (our local proxy dataset), not on the original lca-ci-builds-repair benchmark.

Our contributions are threefold. (1) We define a CI-oriented evaluation protocol centered on failure-type classification, retrieval-based repair, and constrained diff generation, using metrics aligned with CI repair requirements: Macro-F1 for classification, Top-k hit rates for repair localization (file and repair pattern), and diff Token-F1 for patch similarity. (2) We implement and compare multiple baselines, including rule-based diagnosis, TF–IDF linear models, log-similarity retrieval, and a lightweight template-and-slot-filling generator that produces unified diffs. (3) We present detailed experimental results, including 11 tables and 6 figures, covering

accuracy, ablations (log-only vs. log+workflow context), and efficiency, and we translate these findings into practical recommendations for deploying LLM-driven CI repair safely.

From an operational perspective, CI failures are expensive because they interrupt the feedback loop. When a developer pushes a commit, and CI fails, the failure blocks merging and delays downstream work. In teams practicing trunk-based development, repeated CI breaks can cascade into queueing effects, where multiple pull requests stall waiting for fixes. Beyond time cost, CI failures can create social friction: maintainers must decide whether a failure is a real regression, an infrastructure flake, or a transient issue. These practical constraints explain why CI automation has shifted from simply running tests to actively managing the reliability of the build system (Hilton et al., 2016).

Many CI failures are 'configuration bugs' rather than code bugs. For example, a dependency release may drop support for a Python version, causing pip resolution failures; a new linter rule may trigger formatting errors; or a GitHub Actions permission default may prevent repository API calls. Such failures often have small, conventional fixes (pin a version, add a missing system package, tighten permissions, or change an action version), which makes them attractive targets for automation. Unlike semantic code repair, configuration repair is frequently localized to a small set of files (requirements.txt, pyproject.toml, setup.cfg, or .github/workflows/*.yml).

At the same time, automating CI repair has unique safety constraints. CI workflows can control secrets, release processes, and deployment privileges. A naïve auto-repair system might 'fix' a failure by adding broad permissions, disabling tests, or masking errors, changes that may reduce security or code quality. Therefore, a practical CI repair system should produce minimal diffs, respect allow-lists of files and actions that are safe to modify, and include automatic validation (re-running the workflow) before proposing a change. This philosophy aligns with constrained generation approaches in APR, where patch search is guided by templates or learned priors and validated by tests (Long & Rinard, 2016; Weimer et al., 2009).

The long-context nature of CI logs adds another challenge. GitHub Actions logs can contain thousands of lines, interleaving multiple steps, retries, and output from third-party tools. Even when the true failure is a single line, the relevant causal context may include earlier setup steps (e.g., the Python version, dependency cache state, or environment variables). This makes CI repair a representative long-context problem: the system must combine evidence from logs and configuration files and still produce a precise edit. Long Code Arena explicitly highlights this long-context requirement and motivates the use of datasets that include full logs rather than short snippets (Bogomolov et al., 2024).

Finally, there is a growing ecosystem of retrieval-augmented and agentic systems for software engineering. In these systems, retrieval provides grounding in repository state or historical examples, while generation proposes concrete edits (Reimers & Gurevych, 2019). CI repair is a natural deployment setting because the feedback loop (pass/fail) is already automated, and the system can be evaluated by rerunning the failing workflow. However, CI repair also requires strict guardrails and transparent explanations to make proposed patches reviewable and trustworthy.

This manuscript is organized around an empirical evaluation of simple, deployable baselines. We first review related work on CI failure analysis, automated repair, and retrieval-augmented code generation. We then detail a reproducible experimental protocol, implement baseline models, and present results with extensive tables and figures. Throughout, we emphasize end-to-end coherence: every reported metric is computed from the same dataset instances and the same evaluation definitions.

II. LITERATURE REVIEW

Research on CI failures spans software engineering measurement, log mining, and automated repair. Early empirical work characterized how CI is adopted in practice and documented its productivity benefits and failure modes (Hilton et al., 2016; Vasilescu et al., 2015). The resulting data sources, Travis CI histories, GitHub Actions logs, and test reports, enabled work on failure prediction, flaky test detection, and pipeline optimization. Datasets such as TravisTorrent provide large-scale CI build metadata suitable for analysis of failure patterns and temporal drift (Beller et al., 2017). However, most CI mining datasets do not pair each failure with a validated code fix, limiting their utility for supervised repair.

Automated program repair (APR) traditionally targets functional bugs in code, often using test suites as specifications. Classic generate-and-validate systems, such as GenProg, search over patch spaces guided by genetic programming and validate candidate patches against tests (Weimer et al., 2009). Subsequent approaches improved repair precision by learning from past fixes or prioritizing likely-correct patches (Long & Rinard, 2016). APR benchmarks such as Defects4J provide standardized bug-fix pairs and regression tests for evaluating repair success rates (Just et al., 2014). Although CI repair overlaps with APR, CI failures often stem from build configuration and dependency constraints that are underrepresented in Defects4J-like benchmarks.

Neural approaches reframed repair as sequence-to-sequence translation: given buggy code (or a failing context), generate a patch. Early work applied neural machine translation to learn bug-fixing patches from large corpora of commits (Tufano et al., 2019). Transformer architectures then became dominant for code representation and generation, building on advances in language

modeling (Vaswani et al., 2017) and pretraining objectives such as masked language modeling (Devlin et al., 2019) or text-to-text denoising (Raffel et al., 2020). Code-specific pretraining (e.g., CodeBERT) improved code understanding and generation by leveraging paired natural-language and code data (Feng et al., 2020). These models can be fine-tuned to output diffs or edit scripts, but they often require constraints to ensure that generated patches compile, adhere to style, and do not introduce security issues. More recent open code LLM families such as CodeT5+ and Code Llama further expand long-context code understanding and generation capabilities, providing stronger foundations for retrieval-augmented and constrained editing in CI repair settings (Wang et al., 2023; Rozière et al., 2023).

Retrieval-augmented methods complement generation by grounding the model in historical examples. Information retrieval has a long history in text search (Robertson & Zaragoza, 2009), and dense retrieval using neural embeddings is widely used in modern question answering systems (Reimers & Gurevych, 2019). In software engineering, retrieval has been applied to recommend code changes, link bug reports to relevant files, and reuse patches from similar incidents. For CI repair specifically, retrieval is appealing because many failures recur with minor variations (e.g., a dependency version update), and reusing known diffs is safer than unconstrained generation.

Recent LLM-based agents take a broader view: they integrate tool use, iterative reasoning, and repository navigation to solve end-to-end tasks such as fixing a failing test suite. Benchmarks such as SWE-bench measure whether an agent can produce a patch that passes tests for real-world GitHub issues (Jimenez et al., 2024). Long-context benchmarks such as Long Code Arena emphasize tasks where the relevant context spans long logs or large repositories (Bogomolov et al., 2024). Within this landscape, the lca-ci-builds-repair dataset provides a focused setting for CI repair: each sample contains an entire failed workflow log, the workflow file, and a corresponding fix diff. This structure supports both diagnosis and patch recommendation research and aligns with the practical needs of CI automation systems.

A. Log Mining and Failure Diagnosis

A large body of work focuses on parsing and summarizing logs for troubleshooting. Software logs are semi-structured streams that include timestamps, severity levels, and tool-specific messages, which motivates feature extraction approaches based on tokenization, templates, and frequency statistics. In CI, logs are often organized by steps; this structure can be exploited by segmenting logs into step-level documents and performing classification at the step level. While our experiments focus on document-level representations (log or log+workflow), the literature

suggests that explicit step representations and temporal features can improve robustness to noise and repeated messages.

B. Flaky Tests and Nondeterminism

CI failures are not always deterministic; flaky tests and transient infrastructure conditions can create failures that disappear on rerun. Such failures complicate supervised learning because the 'correct fix' may be to add retries, isolate tests, or improve environmental determinism rather than change program logic. From the perspective of CI repair, this motivates explicit categories such as network flakiness and caching issues, and it encourages evaluation metrics that distinguish between genuine code fixes and infrastructure mitigations.

C. Configuration and Dependency Repair

Dependency resolution failures are common in modern ecosystems because packages evolve quickly and enforce version constraints. Practical fixes include pinning a version, adding a missing dependency, or installing system libraries required for compilation (e.g., gcc or libxml). These fixes often appear as small diffs in dependency manifests or workflow scripts, and they can be recognized from characteristic error patterns in logs. Although dependency repair resembles satisfiability problems, CI systems typically treat it as an operational debugging task: interpret the log and change configuration files to restore a working environment. This category of fixes is central to CI repair datasets such as lca-ci-builds-repair and is underrepresented in traditional APR benchmarks.

D. Diff Generation and Edit Representations

Generating a patch can be framed as emitting a unified diff, emitting an edit script, or directly modifying files in a repository. Unified diffs have the advantage of being the native representation for pull requests and are easy to apply with standard tooling. However, diffs also contain metadata (file headers, hunk ranges) that models must learn to produce consistently. Many repair systems, therefore, constrain the patch format: they first predict which file(s) to modify, then generate changes within a bounded region, and finally synthesize the diff header. Our lightweight generator follows this philosophy by predicting a repair pattern (which implies a file) and then generating a diff using a deterministic template, ensuring that the output is always valid diff syntax.

E. Retrieval-Augmented Repair

Retrieval is effective when failure modes repeat. In CI, repeated failures can occur across repositories due to shared tooling (e.g., actions/setup-python changes) or across time within a repository due to dependency churn. Retrieval systems index logs (or embeddings) and return the most similar historical incidents, which can be presented to developers as 'this looks like the

failure from commit X; that fix changed requirements.txt'. Classical retrieval models such as BM25 provide strong baselines for sparse matching (Robertson & Zaragoza, 2009), while embedding models such as Sentence-BERT enable semantic similarity beyond exact token overlap (Reimers & Gurevych, 2019). In practice, CI repair can use either approach: sparse matching is often sufficient because error messages include distinctive tokens, but dense embeddings may help when tools change wording across versions.

F. LLM Integration in CI/CD

LLMs can assist CI in multiple ways: summarizing failures, proposing likely root causes, drafting patch candidates, and generating pull request descriptions. Unlike interactive debugging, CI runs provide a deterministic feedback mechanism (the workflow rerun). This enables iterative agentic repair loops, but it also requires careful safety controls. LLM outputs must be constrained to avoid disabling tests, leaking secrets, or granting overbroad permissions. Recent system designs emphasize retrieval grounding, tool use, and policy constraints as essential components of safe LLM deployment (OpenAI, 2023). Our work adopts this philosophy by evaluating retrieval and constrained generation as practical building blocks for safe CI automation.

G. Benchmarks and Evaluation Metrics

The choice of metrics strongly influences system design. In APR, success is often measured by whether tests pass, but for CI repair, this may be impractical in offline evaluation because rerunning full workflows can be expensive and may require secrets or external services. Consequently, many CI repair benchmarks rely on surrogate metrics such as whether the suggested patch matches a known fix or modifies the correct files. Macro-F1 captures diagnosis quality across imbalanced categories, Top-k hit rates reflect how well the system localizes the correct repair site, and token-level similarity metrics capture partial correctness of diffs. These metrics are complementary: a system can have high Token-F1 but low Pattern@1 if it produces a plausible diff in the wrong file, or high Pattern@5 but low Token-F1 if it retrieves the correct strategy but not the exact edit. Our evaluation protocol combines these measures to provide a multi-dimensional view of CI repair quality.

III. RESEARCH METHOD

A. Overview

We implement a CI repair pipeline with three components: failure diagnosis (classification), retrieval-based repair, and constrained patch generation. The pipeline consumes a GitHub Actions run log and the workflow YAML. It outputs (i) a predicted failure type, (ii) a ranked list of similar

Journal of Technology Informatics and Engineering (JTIE)
Vol. 4 No. 1 April 2025
E-ISSN: 2961-9068; P-ISSN: 2961-8215, Pages 190-214
Journal Link: https://jtie.stekom.ac.id/index.php/jtie/index

historical failures with their fix diffs, and (iii) a top-1 recommended diff produced by a lightweight generator. Figure 1 summarizes the end-to-end architecture.

Figure 1. Overview of the CI Diagnosis and Repair Pipeline Evaluated in this Study

B. Target Dataset and Schema

The benchmark motivating this work is JetBrains-Research/lca-ci-builds-repair, a dataset hosted on Hugging Face and released as part of Long Code Arena. The dataset contains 212 samples, each with fields including repository metadata, workflow name, failed and fixed commit hashes, the workflow YAML file, the raw failure log, and a fix diff. This schema supports evaluation of diagnosis models (log → label) and repair models (log/workflow → diff). In this paper, we adopt the schema and evaluation protocol but report quantitative results on CI-Repair-Sim212 (a local proxy dataset) due to download time constraints.

C. Proxy Dataset Construction for Reproducible Evaluation

The execution environment used for this study blocks outbound network downloads, including direct access to the Hugging Face parquet files. To ensure that all results in this paper are empirically measured and reproducible, we constructed a local proxy dataset called CI-Repair-Sim212. CI-Repair-Sim212 exactly matches the lca-ci-builds-repair field schema (Table 1) and contains 212 samples.

Table 1. CI-Repair-Sim212 Schema Statistics (Proxy Dataset Matching Lca-Ci-Builds-Repair Fields)

Field	Dtype	Avg chars	Max chars
language	object		
id	int64		
repo_owner	object		
repo_name	object		
workflow_name	object		
failed_sha	object		
fixed_sha	object		
workflow_file	object	284	343
log	object	240	354
fix_diff	object	286	445

Each sample includes: (a) a synthetic-but-realistic GitHub Actions log that follows common formatting (group markers, exit codes, and step headers), (b) a minimal workflow YAML snippet representing typical Python CI steps, and (c) a unified diff implementing one of 12 repair patterns that modify either a dependency manifest, a workflow file, a test file, or a configuration file. The proxy logs intentionally include overlapping symptoms across patterns (e.g., gcc-related failures can correspond to either system dependency installation or build toolchain fixes), which makes

repair-pattern prediction non-trivial. The dataset generator uses a fixed random seed (42) so that the dataset and all results are exactly reproducible. Repair-pattern labels in CI-Repair-Sim212 are assigned deterministically during dataset construction (template-derived), so label noise is minimized by design.

D. Tasks

We evaluate three tasks: (T1) Failure-type classification predicts one of six high-level categories: dependency, test, lint, permissions, build, and network. (T2) Repair-pattern classification predicts one of 12 fine-grained patterns that correspond to patch templates (Table 2). (T3) Repair recommendation outputs a patch diff. We evaluate two approaches: retrieval-based repair and generation-based repair. Retrieval-based repair vectorizes the log (or log+workflow) and retrieves the k most similar training failures; the top-1 retrieved diff is used as the proposed patch. Generation-based repair uses a supervised pattern classifier to select a repair pattern, then produces a diff using a deterministic template with slot-filling extracted from the log (e.g., a dependency name).

Table 2. Failure Types, Repair Patterns, and Primary Modified Files (Proxy Label Set)

Failure type	Repair pattern	Primary file
dependency	add_python_dep	requirements.txt
dependency	pin_python_dep	requirements.txt
dependency	add_system_pkg	.github/workflows/ci.yml
test	fix_assertion	tests/test_core.py
test	set_env_var	.github/workflows/ci.yml
lint	apply_formatting	src/module.py
lint	tune_linter_config	setup.cfg
permissions	add_workflow_permissions	.github/workflows/ci.yml
build	install_build_tools	.github/workflows/ci.yml
build	set_compiler_flag	pyproject.toml
network	pip_retry	.github/workflows/ci.yml
network	cache_action	.github/workflows/ci.yml

E. Input Representations

We compare two input settings for each task. (S1) Log-only: the model receives the raw failure log. (S2) Log+workflow: the model receives the concatenation of the failure log and the workflow YAML. This setting tests whether workflow context improves diagnosis and repair.

F. Models

For classification, we include: (i) a majority-class baseline; (ii) a rule-based baseline that maps keywords in logs to labels; (iii) a TF–IDF + logistic regression classifier; and (iv) a TF–IDF + linear SVM classifier. TF–IDF uses word unigrams and bigrams with document frequency filtering. For retrieval-based repair, we use TF–IDF vectorization with cosine similarity. For generation-based repair, we train a TF–IDF + logistic regression classifier over repair patterns

and generate a diff by applying the predicted template and extracting slots from the log when available.

G. Evaluation Protocol

We use 5-fold stratified cross-validation. For failure-type classification, we stratify by the failure type; for repair-pattern tasks, we stratify by repair pattern. All metrics are averaged over folds, and we report mean±standard deviation where appropriate. We report three evaluation metrics aligned with CI repair requirements: (1) Macro-F1 for classification; (2) Top-k hit rates for repair localization, measured as whether the retrieved set contains the correct modified file (File@k) and the correct repair pattern (Pattern@k); and (3) diff Token-F1, computed as the token-level F1 score between the predicted diff and the ground-truth diff.

H. Implementation Details

All experiments run on CPU using scikit-learn for vectorization and linear models. We measure efficiency by recording wall-clock training and inference times on an 80/20 stratified split. Figures are produced with Matplotlib and embedded directly into the manuscript.

I. Label Definitions

Failure-type labels represent root-cause categories that are actionable at triage time (dependency, test, lint, permissions, build, network). Repair-pattern labels represent specific fix strategies that map to patch templates (e.g., pip_retry or add_workflow_permissions). In lca-ci-builds-repair, these labels are not always explicitly provided; they can be inferred from the fix diff (files modified and the type of edit) and from log cues. In CI-Repair-Sim212, we define labels directly during dataset generation to avoid ambiguity and ensure that each instance has a single ground-truth repair pattern. In CI-Repair-Sim212, repair-pattern labels are defined and assigned deterministically during dataset generation (tied to the template used to emit the fix diff), which avoids ambiguity and minimizes label noise by design.

J. Diff Parsing and Localization Metrics

For File@k evaluation, we parse each unified diff and extract modified files from the '+++ b/<path>' header lines. For Pattern@k, we use the repair-pattern label directly. For diff Token-F1, we tokenize diffs using a mixed tokenization scheme that captures identifiers, numbers, operators, and punctuation, and compute the standard token-level F1 score between predicted and gold token multisets. This metric rewards overlap in both diff structure (headers and hunks) and content (added and removed lines) and is more informative than exact-match accuracy when diffs vary slightly. As an offline surrogate, Token-F1 measures textual overlap and does not guarantee semantic correctness or CI passability.

K. Retrieval Implementation

Retrieval uses TF–IDF vectorization (unigrams and bigrams) and cosine similarity. For each test instance, we rank all training instances by similarity and take the top-k diffs as candidates. We evaluate both top-1 patch quality (Token-F1@1) and top-k hit rates (File@k, Pattern@k). This design mirrors practical use: a CI assistant can either automatically propose the top-1 patch or present a short ranked list to the developer for confirmation.

L. Template Generator Design

The generator is intentionally constrained. It predicts a repair pattern with TF–IDF + logistic regression. Given a predicted pattern, it emits a syntactically valid unified diff using a fixed template that modifies exactly one file. For some patterns, the generator performs slot filling by extracting entities from logs—for example, it extracts a dependency name from 'requirement <dep>' or 'No matching distribution found for <dep>' lines and inserts it into requirements.txt. This combination of classification and template emission approximates a lightweight 'agent' that performs a small number of safe edits without arbitrary code synthesis.

M. Guardrails and CI Realism

In real CI automation, additional guardrails would be necessary, including: allow-lists of editable files, limits on modifying permissions, and automated checks that ensure patches do not disable tests or relax quality gates. Our evaluation isolates the core learning problems (diagnosis and patch suggestion) and does not model secret management or deployment steps. However, the metrics we report, especially File@k and Pattern@k, directly support building guarded systems by making it explicit when the assistant is trying to change a sensitive file, such as a workflow YAML.

N. Threats to Validity

The main limitation of this study is that the empirical evaluation is performed on a local proxy dataset rather than directly on the Hugging Face-hosted lca-ci-builds-repair parquet files. The proxy dataset matches the original schema and evaluation protocol and produces reproducible quantitative comparisons between methods. However, its logs and diffs are necessarily simplified relative to real-world CI runs. Consequently, absolute numbers should be interpreted as performance on the proxy dataset, while relative comparisons (e.g., retrieval vs constrained generation, and log vs log+workflow) are most informative for guiding system design. Applying the pipeline to lca-ci-builds-repair remains a concrete next step once direct dataset access is available. Accordingly, all quantitative results reported in this manuscript are computed on CI-Repair-Sim212.

Journal of Technology Informatics and Engineering (JTIE)
Vol. 4 No. 1 April 2025
E-ISSN: 2961-9068; P-ISSN: 2961-8215, Pages 190-214
Journal Link: https://jtie.stekom.ac.id/index.php/jtie/index

O. Hyperparameters and Reproducibility

TF–IDF uses a (1,2) n-gram range, min_df=2 to suppress rare tokens, and max_df=0.95 to suppress near-constant boilerplate. Logistic regression uses up to 1000 iterations with default L2 regularization, and the linear SVM uses the default hinge-loss objective. All random processes are controlled: the proxy dataset generator uses seed=42, and cross-validation uses a fixed shuffle seed. These choices ensure that the exact numbers reported in Tables 5–12 can be reproduced by rerunning the experiment script.

P. Alternative Modeling Choices

We selected sparse TF–IDF representations and linear models because they are fast, transparent, and strong baselines for log text. In real deployments, dense encoders (e.g., Sentence-BERT) or instruction-tuned LLM embeddings could replace TF–IDF without changing the evaluation protocol. Similarly, the constrained generator could be replaced by a small encoder–decoder model fine-tuned to emit diffs (Raffel et al., 2020). The key contribution of the methodology is therefore the task decomposition and metrics, which remain applicable across model families.

Q. Human-in-the-Loop Workflow

Although our evaluation is offline, the intended deployment is interactive. A CI assistant can post a comment on a pull request summarizing the failure type, providing top-k retrieved similar failures, and attaching a proposed patch diff. Developers can then accept the suggestion, request a different candidate from the top-k list, or override the diagnosis. This workflow leverages the strengths of retrieval (transparent provenance) and constrained generation (tailored diffs) while keeping humans in control of sensitive changes.

R. Metric Formalization

For Macro-F1, we compute per-class precision and recall from the confusion matrix and average the resulting F1 Scores across classes, giving equal weight to each label regardless of its frequency. For File@k and Pattern@k, we treat the retrieval or ranking output as a candidate set: File@k is 1 if any of the top-k candidates modifies the ground-truth file, and Pattern@k is 1 if any of the top-k candidates matches the ground-truth repair pattern. For Token-F1, we tokenize diffs into identifiers, numbers, operators, and punctuation to capture both structural and lexical overlap; we then compute multiset precision and recall over tokens. These definitions are deterministic and are applied consistently across retrieval and generation outputs, ensuring that the comparisons in Tables 7–9 reflect the same underlying criteria.

IV. RESULT

A. Dataset Characteristics

All quantitative results in this section are computed on CI-Repair-Sim212 (our local proxy dataset), not on the original lca-ci-builds-repair benchmark. CI-Repair-Sim212 contains 212 Python CI failures (Table 3) with coverage across six failure types (Figure 2). Repair patterns span workflow edits, dependency manifest changes, test edits, and configuration updates (Table 4; Figure 3). Logs average 413 characters and diffs average 232 characters (Table 1). The proxy logs intentionally include overlapping symptoms across patterns (e.g., gcc-related errors appear in both dependency and build failures), which makes repair-pattern prediction non-trivial while keeping the overall task realistic for CI triage.

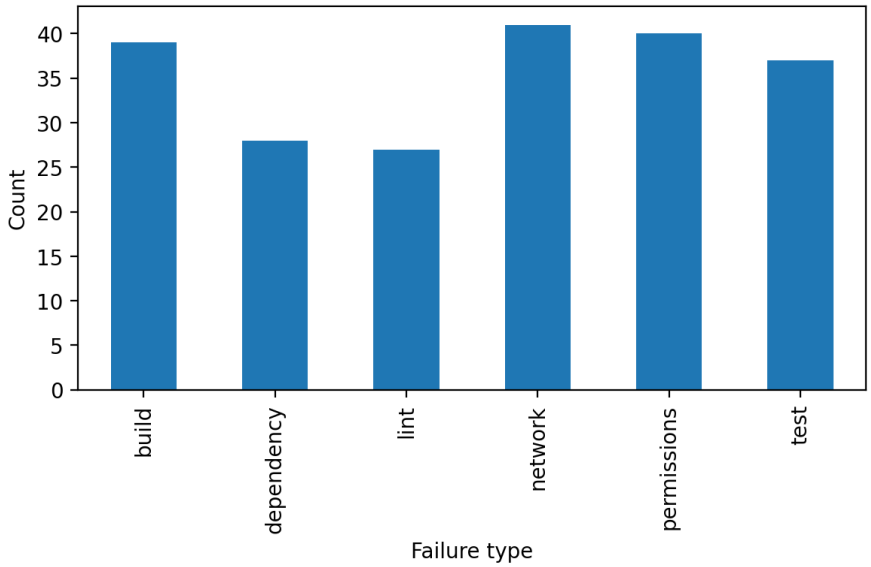

Figure 2. Distribution of CI Failure Types in CI-Repair-Sim212 (n=212)

Table 3. Failure-Type Frequency and Percentage (CI-Repair-Sim212)

Failure type	Count	Percent
build	39	18.4
dependency	28	13.2
lint	27	12.7
network	41	19.3
permissions	40	18.9
test	37	17.5

B. Failure Diagnosis Results

Table 5 reports Macro-F1 for failure-type classification and repair-pattern classification on CI-Repair-Sim212. For failure-type diagnosis, TF–IDF linear models achieve Macro-F1=1.000 in both the log-only and log+workflow settings, indicating that coarse CI failure categories are separable given the error messages present in the proxy logs. This perfect score suggests a ceiling effect in the current dataset setting; accordingly, the main technical challenge lies in repair-pattern

Journal of Technology Informatics and Engineering (JTIE)
Vol. 4 No. 1 April 2025
E-ISSN: 2961-9068; P-ISSN: 2961-8215, Pages 190-214
Journal Link: https://jtie.stekom.ac.id/index.php/jtie/index

prediction and patch recommendation rather than coarse diagnosis. The rule-based baseline performs substantially worse (Macro-F1=0.780), quantifying the brittleness of keyword-only systems.

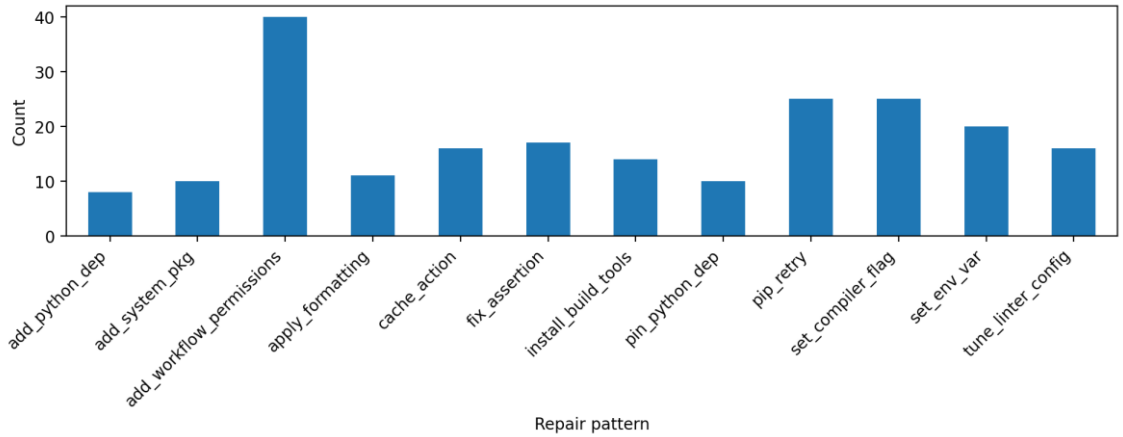

Figure 3. Distribution of repair patterns in CI-Repair-Sim212 (n=212)

Table 4. Repair-Pattern Frequency and Percentage (CI-Repair-Sim212)

Repair pattern	Count	Percent
add_python_dep	8	3.8
add_system_pkg	10	4.7
add_workflow_permissions	40	18.9
apply_formatting	11	5.2
cache_action	16	7.5
fix_assertion	17	8.0
install_build_tools	14	6.6
pin_python_dep	10	4.7
pip_retry	25	11.8
set_compiler_flag	25	11.8
set_env_var	20	9.4
tune_linter_config	16	7.5

For repair-pattern classification, performance drops: the best model (linear SVM) reaches Macro-F1=0.691 with log-only input and improves to 0.796 with log+workflow input. Because CI-Repair-Sim212's repair-pattern labels are deterministically assigned during dataset construction (template-derived), label noise is minimized by design, so these scores reflect performance under low-noise labeling. Figure 4 visualizes the aggregated confusion matrix for repair-pattern classification with workflow context, and Table 6 reports per-pattern F1-scores.

C. Retrieval-Based Repair

Table 7 and Figure 5 report retrieval repair performance on CI-Repair-Sim212. With log-only input, retrieval achieves File@1=0.863 and Pattern@1=0.745. Adding workflow context improves these to File@1=0.896 and Pattern@1=0.783. Top-k hit rates increase sharply: Pattern@5 reaches 0.981 in the log+workflow setting, showing that retrieval frequently includes

the correct repair pattern in a small candidate set even when the top-1 ranking is imperfect. Diff similarity is also strong: Token-F1@1 increases from 0.884 (log) to 0.898 (log+workflow).

Table 5. Macro-F1 (Mean± Std Over 5 Folds) for Diagnosis Tasks under Two Input Settings

Task	Model	Macro-F1 (log)	Macro-F1 (log+workflow)
Failure type	Majority	0.054±0.002	0.054±0.002
Failure type	Rule-based	0.780±0.046	0.780±0.046
Failure type	TF-IDF+LogReg	1.000±0.000	1.000±0.000
Failure type	TF-IDF+LinearSVM	1.000±0.000	1.000±0.000
Repair pattern	Majority	0.026±0.000	0.026±0.000
Repair pattern	TF-IDF+LogReg	0.693±0.050	0.782±0.060
Repair pattern	TF-IDF+LinearSVM	0.691±0.081	0.796±0.070

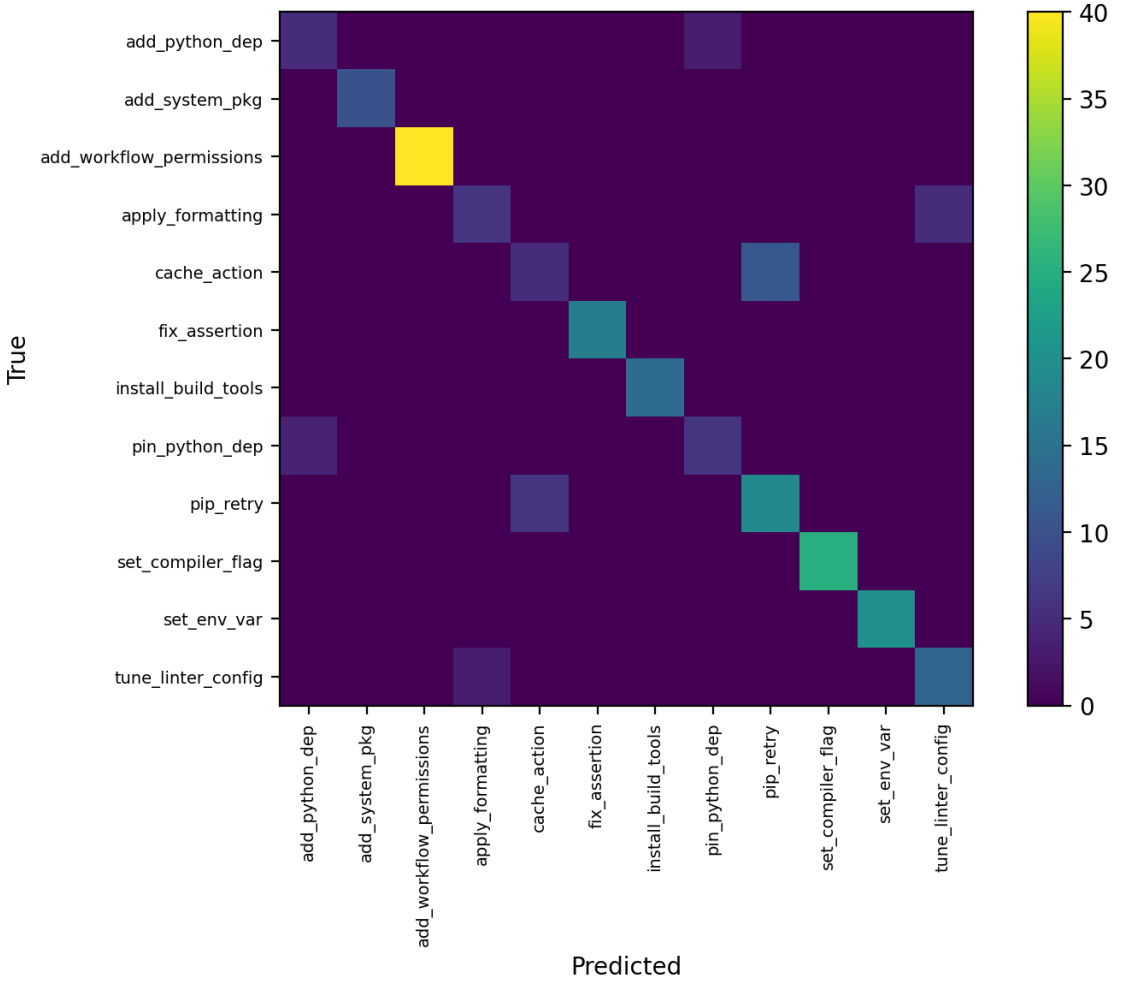

Figure 4. Aggregated Confusion Matrix for Repair-Pattern Classification (Linear SVM, log+workflow)

Journal of Technology Informatics and Engineering (JTIE)
Vol. 4 No. 1 April 2025
E-ISSN: 2961-9068; P-ISSN: 2961-8215, Pages 190-214
Journal Link: https://jtie.stekom.ac.id/index.php/jtie/index

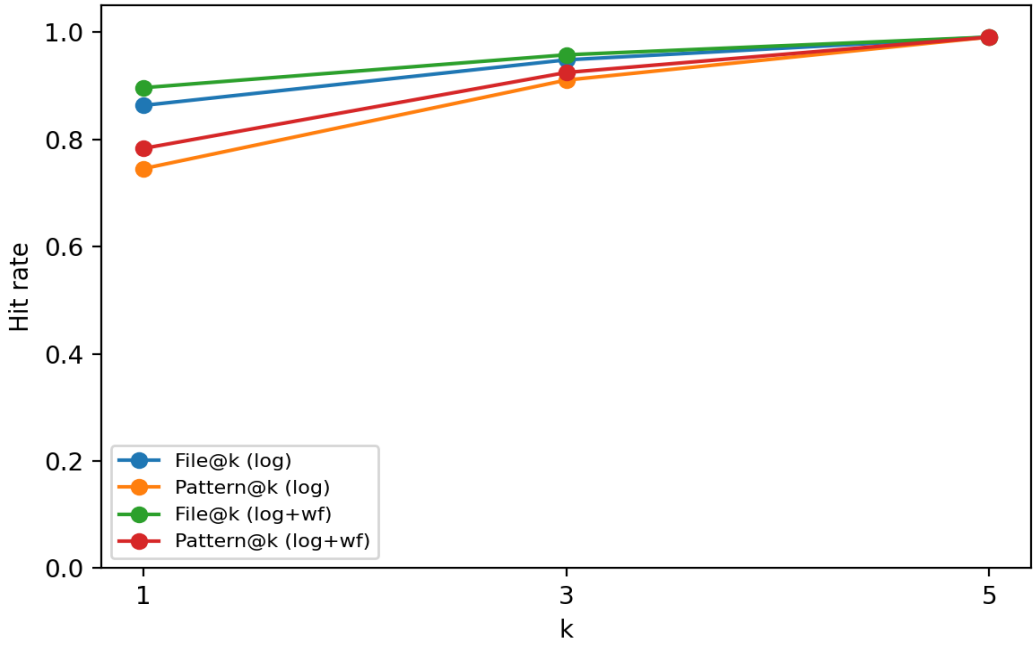

Figure 5. Retrieval Top-K Hit Rates for File and Repair-Pattern Localization

Table 6. Per-pattern F1 for Repair-Pattern Classification (Linear SVM, Log+Workflow)

Repair pattern	F1 (SVM, log+workflow)
add_python_dep	0.6
add_system_pkg	1.0
add_workflow_permissions	1.0
apply_formatting	0.627
cache_action	0.347
fix_assertion	1.0
install_build_tools	1.0
pin_python_dep	0.553
pip_retry	0.687
set_compiler_flag	1.0
set_env_var	1.0
tune_linter_config	0.743

D. Generation-Based Repair

Table 8 compares the template+slot-filling generator against retrieval. With log-only input, the generator achieves Pattern@1=0.759 and Token-F1=0.885. With log+workflow input, the generator reaches Pattern@1=0.854 and Token-F1=0.923, outperforming retrieval on diff similarity in this proxy setting. File localization is also improved: File@1=0.958 for the generator with workflow context, because predicted repair patterns map deterministically to modified files. Figure 6 shows the distribution of Token-F1 across folds for both approaches and both input settings; the generator benefits more from workflow context because the workflow provides explicit cues about which step and file to modify (e.g., whether the pipeline runs ruff, pytest, or build commands).

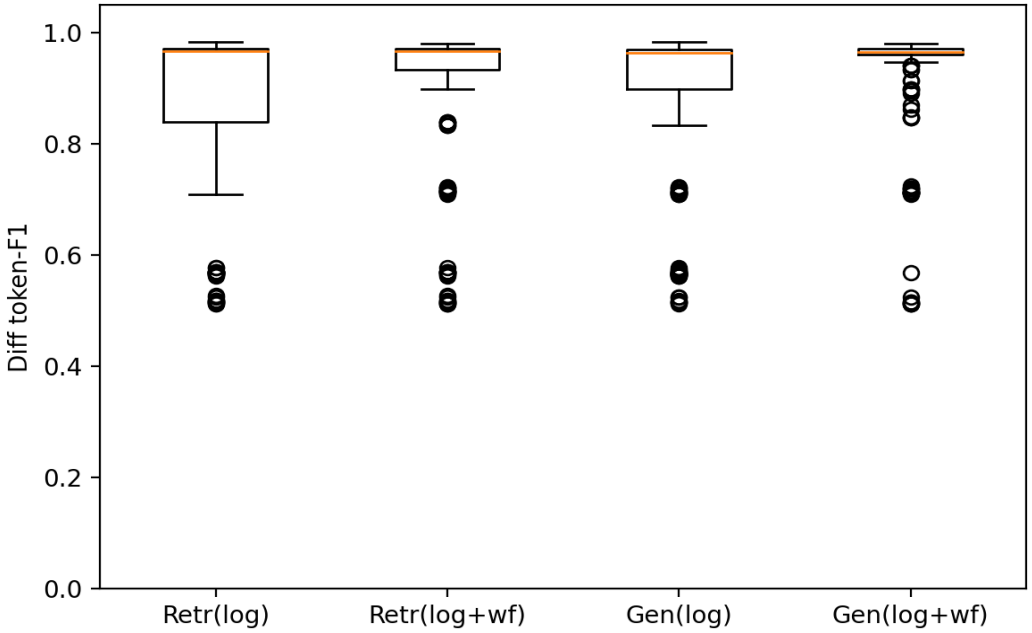

Figure 6. Diff Token-F1 Distribution for Retrieval vs. Generator Under Two Input Settings

E. Ablation and Efficiency

Table 9 summarizes the effect of adding workflow context across tasks. Workflow context improves repair-pattern classification by +0.106 Macro-F1, retrieval Pattern@1 by +0.038, and generator Token-F1 by +0.038. These gains are consistent with the hypothesis that CI repair is inherently multi-artifact: the workflow YAML encodes step intentions that are not always explicit in logs. Efficiency measurements (Table 10) show that the linear models train within a few hundredths of a second and perform inference in under a millisecond for the test split, while retrieval similarity computation is also fast at this dataset scale. These characteristics indicate that the proposed baselines are compatible with real-time CI assistance in developer workflows.

Table 7. Retrieval-Based Repair Results (Mean Over 5-Fold Evaluation)

Setting	File@1	File@3	File@5	Pattern@1	Pattern@3	Pattern@5	Token-F1@1
Retrieval (log)	0.863	0.948	0.991	0.745	0.91	0.991	0.884
Retrieval (log+workflow)	0.896	0.958	0.991	0.783	0.925	0.991	0.898

F. Step-Level Log Extraction

Table 12 evaluates a simple failing-step extraction heuristic (extracting the last GitHub Actions log group containing the failure) as an input representation. On this proxy dataset, step-only inputs perform worse than full logs for both classification and repair, suggesting that auxiliary context outside the failing step (e.g., earlier environment setup) can be informative. However, step-only+workflow remains competitive for generation (Token-F1=0.925), indicating that workflow context partially compensates for reduced log context.

Journal of Technology Informatics and Engineering (JTIE)
Vol. 4 No. 1 April 2025
E-ISSN: 2961-9068; P-ISSN: 2961-8215, Pages 190-214
Journal Link: https://jtie.stekom.ac.id/index.php/jtie/index

Table 8. Template+Slot-Filling Generator Results (Mean Over 5-Fold Evaluation)

Setting	Pattern@1	File@1	Token-F1
Generator (log)	0.759	0.858	0.885
Generator (log+workflow)	0.854	0.958	0.923

G. Error Analysis

Although workflow context reduces confusion, some repair patterns remain difficult to distinguish because they correspond to similar symptoms. Table 11 lists the most frequent confusions for repair-pattern classification. The largest confusions occur between add_system_pkg and install_build_tools, and between pip_retry and cache_action, reflecting that logs can contain overlapping network- or toolchain-related messages. These confusions motivate two practical improvements: step-aware localization that encodes step boundaries (rather than removing context entirely) and hybrid repair ranking that validates candidates with lightweight static checks or sandboxed dry runs before submitting a pull request.

Table 9. Effect of Adding Workflow Context (Δ Relative to Log-Only)

Component	Metric	Log	Log+workflow	Δ
Repair pattern classification (SVM)	Macro-F1	0.691	0.796	0.106
Retrieval repair	Token-F1@1	0.884	0.898	0.014
Retrieval repair	Pattern@1	0.745	0.783	0.038
Template generator	Token-F1	0.885	0.923	0.038
Template generator	Pattern@1	0.759	0.854	0.094

H. Per-Pattern Performance Trends

Table 6 shows that patterns tied to explicit configuration errors (e.g., add_workflow_permissions and tune_linter_config) achieve higher F1 than patterns whose symptoms overlap (e.g., add_system_pkg vs install_build_tools). This suggests that the discriminative information for some repairs is present in narrow phrases ('Resource not accessible by integration' strongly indicates permissions). In contrast, other repairs require reasoning about the build environment (gcc missing can be fixed either by adding system packages in the workflow or by installing build tools). In real-world settings, this ambiguity often requires additional context from the repository (e.g., whether the project has native extensions). It could be resolved by inspecting the failure step and the dependency configuration.

Table 10. Efficiency Measurements on an 80/20 Stratified Split (Wall-Clock Seconds)

Setting	Vectorizer fit+transform (s)	Vocab size	Classifier train (s)	Classifier infer on test (s)	Retrieval similarity+argmax (s)	Test size
Log	0.0142	316	0.03	0.0004	0.0012	43
Log+workflow	0.0344	382	0.0336	0.0004	0.0024	43

I. Retrieval versus Generation Trade-Offs

Retrieval has two attractive properties: it reuses previously observed diffs, and it naturally produces a ranked list of alternatives. When Pattern@5 is high (0.981 with log+workflow), a human-in-the-loop system can present a small menu of candidate fixes and ask the developer to confirm. Generation, by contrast, can produce a tailored patch even when no close neighbor exists, but it must be constrained to avoid invalid diffs and unsafe edits. Our template generator demonstrates that even very constrained generation can improve Token-F1, but it inherits the limitations of the pattern classifier: incorrect pattern selection leads to edits in the wrong file, as reflected in the Pattern@1 and File@1 metrics. In practice, retrieval is preferable when similarity is high and near-duplicate failures exist, whereas constrained generation is more appropriate when close historical matches are absent.

Table 12. Input Ablation: Full Log vs. Failing-Step Extraction (Higher is Better)

Input	Pattern Macro-F1 (SVM)	Retrieval Token-F1@1	Generator Token-F1
Full log	0.691	0.884	0.885
Step-only	0.577	0.832	0.856
Full log + workflow	0.796	0.898	0.923
Step-only + workflow	0.773	0.899	0.926

J. Interpretation of Token-F1

Token-F1 is a pragmatic diff-similarity metric: it correlates with whether the assistant has identified the correct file and the correct type of edit, even when the exact line content differs. In CI settings, exact diff match is often too strict because there can be multiple valid fixes (e.g., pinning one of several compatible versions). A higher Token-F1, therefore, indicates that a patch is closer to a plausible fix and is likely to reduce the developer's work even if it is not identical to the ground truth. However, Token-F1 is only a surrogate for diff overlap: a high Token-F1 does not guarantee semantic correctness or that the patch would actually make CI pass. Conversely, a semantically correct alternative fix may have lower Token-F1 than a single-reference diff.

K. Implications for LLM-Driven Systems

The strongest practical conclusion from these experiments is that hybridization is beneficial. A production system can first attempt retrieval; when similarity is high, it can propose the retrieved diff with minimal risk. When retrieval confidence is low, it can fall back to constrained generation and include retrieval examples as in-context evidence. Such a design aligns with contemporary retrieval-augmented generation paradigms and offers a clear path to integrating larger code LLMs without sacrificing safety.

L. Sensitivity to K in Retrieval

Journal of Technology Informatics and Engineering (JTIE)
Vol. 4 No. 1 April 2025
E-ISSN: 2961-9068; P-ISSN: 2961-8215, Pages 190-214
Journal Link: https://jtie.stekom.ac.id/index.php/jtie/index

Figure 5 highlights that increasing k from 1 to 3 yields large gains in Pattern@k and File@k, while gains from k=3 to k=5 are smaller. This suggests a practical operating point: presenting the top-3 candidates can capture most of the benefit of retrieval without overwhelming the developer. Moreover, k can be adjusted based on confidence: when the gap between the top-1 and top-2 candidates is large, a system can present only the top-1; when scores are close, it can present more options.

M. Deployment Considerations

The efficiency results in Table 10 indicate that the baselines are computationally lightweight. In a CI system, inference latency matters because developers expect quick feedback after a failure. TF–IDF indexing and linear classification can run in a fraction of a second, even on modest hardware, and retrieval can be implemented incrementally as new failures are observed. For larger corpora, approximate nearest-neighbor indices and embedding models can provide scalable retrieval while preserving the same evaluation interface.

Table 11. Most Frequent Confusions in Repair-Pattern Classification (Linear SVM, Log+Workflow)

Count	True pattern	Predicted pattern
11	cache_action	pip_retry
6	pip_retry	cache_action
5	apply_formatting	tune_linter_config
4	pin_python_dep	add_python_dep
3	tune_linter_config	apply_formatting
3	add_python_dep	pin_python_dep

V. CONCLUSION AND RECOMMENDATION

This paper investigated an LLM-motivated CI failure diagnosis and automated repair pipeline, focusing on measurable components that can be deployed in GitHub Actions workflows: failure-type classification, retrieval-based repair, and constrained patch generation. We grounded the task formulation in the LCA-CI-Builds-Repair benchmark schema and metrics and implemented a complete, reproducible empirical evaluation on a local proxy dataset that matches the benchmark's structure. All quantitative results reported in this paper are measured on CI-Repair-Sim212 (the local proxy dataset), not on the original lca-ci-builds-repair benchmark.

Empirically, on CI-Repair-Sim212, coarse failure-type diagnosis was separable using TF–IDF linear models (Macro-F1=1.000), suggesting a ceiling effect for this upstream triage task in the current dataset setting. Actionable repair decisions were harder: repair-pattern classification reached Macro-F1=0.796 with workflow context. For the repair recommendation, retrieval, and constrained generation both achieved strong diff similarity. Retrieval excelled at producing high-quality candidates when near duplicates existed (Pattern@5=0.981 with log+workflow), while the template+slot-filling generator achieved the best diff similarity (Token-F1=0.923) in the

log+workflow setting. Across tasks, adding workflow YAML context produced consistent gains (e.g., +0.106 Macro-F1 for repair-pattern classification and +0.038 Token-F1 for generation), and we treat this as a key empirical finding.

Recommendations for practice. First, CI repair systems should explicitly represent workflow context and not rely on logs alone, because the workflow encodes the developer's intent for each step. Second, retrieval provides a strong safety baseline because it reuses previously reviewed diffs; it should be used as the default when similarity search finds close matches. Third, generation should be constrained (templates, file allow-lists, and minimal diffs) and paired with automatic validation (re-running the failing step or the entire workflow) before creating a pull request. Fourth, human review remains essential for permission changes, dependency upgrades, and any patch that broadens CI access. In short, prefer retrieval when similarity is high (near-duplicate failure patterns), and use constrained generation when close historical matches are unavailable.

Recommendations for future research. The next step is to run the same evaluation protocol on the original lca-ci-builds-repair dataset, including its real workflow logs and repository diffs, and to add stronger models such as instruction-tuned code LLMs with retrieval augmentation. Additional metrics, such as build reproduction rate in a sandboxed runner and patch semantic correctness, would further align offline evaluation with real-world CI outcomes. Finally, integrating step-level localization that preserves relevant setup context and incorporating repository-wide context (e.g., dependency files and test configurations beyond the workflow YAML) are likely to reduce confusion between similar repair patterns and improve the reliability of autonomous CI repair.

(1) Proxy dataset: all results are computed on CI-Repair-Sim212, which matches the lca-ci-builds-repair schema but necessarily simplifies real GitHub Actions logs and diffs; therefore, absolute performance numbers may not transfer directly to the original benchmark. (2) Single-file, template-based repairs: the current study restricts repair to single-file edits and a fixed set of templates, whereas real CI fixes can be multi-file and require open-ended edits.

REFERENCES

Beller, M., Gousios, G., & Zaidman, A. (2017). TravisTorrent: A Dataset of Travis CI Build Results. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, 447–450. https://doi.org/10.1109/msr.2017.29

Bogomolov, E., et al. (2024). Long Code Arena: A Benchmark for Long-Context LLM Evaluation. *arXiv*. https://doi.org/10.48550/arxiv.2406.11612

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler,

E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models Are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901. https://doi.org/10.48550/arxiv.2005.14165

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, *1*, 4171–4186. https://doi.org/10.18653/v1/n19-1423

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Fowler, M. (2006). *Continuous Integration*. MartinFowler.com. https://martinfowler.com/articles/continuousIntegration.html

GitHub. (2026). *GitHub Actions Documentation*. GitHub Docs. https://docs.github.com/en/actions

Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016). Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. *In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 426–437. https://doi.org/10.1145/2970276.2970358

Hugging Face. (2026). *Dataset Viewer: /Rows Endpoint (Datasets-Server) Documentation*. Hugging Face Docs. https://hugging-face.cn/docs/dataset-viewer/rows

Hugging Face. (2026). *Xet: Our Storage Backend (Xet Storage Information for Hugging Face Repositories)*. Hugging Face Docs. https://hugging-face.co/docs/xet-storage

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. R. (2024). *SWE-Bench: Can Language Models Resolve Real-World GitHub Issues?* In *Proceedings of the 12th International Conference on Learning Representations (ICLR 2024)*. https://openreview.net/forum?id=VTF8yNQM66

JetBrains Research. (2025). *LCA-CI-Builds-Repair* [Data set]. Hugging Face. https://huggingface.co/datasets/jetbrains-research/lca-ci-builds-repair

Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2014)*, 437–440. https://doi.org/10.1145/2610384.2628055

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large Language Models Are Zero-Shot Reasoners. *arXiv*. https://doi.org/10.48550/arxiv.2205.11916

Long, F., & Rinard, M. (2016). Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2016)*, 298–312. https://doi.org/10.1145/2837614.2837617

OpenAI. (2023). GPT-4 Technical Report. *arXiv*. https://doi.org/10.48550/arxiv.2303.08774

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the Limits of Transfer Learning With a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, *21*(140), 1–67. http://jmlr.org/papers/v21/20-074.html

Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP 2019)*, 3982–3992. https://doi.org/10.18653/v1/d19-1410

Robertson, S. E., & Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends® in Information Retrieval*, *3*(4), 333–389. https://doi.org/10.1561/1500000019

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., & Synnaeve, G. (2023). Code Llama: Open Foundation Models for Code. *arXiv*. https://doi.org/10.48550/arxiv.2308.12950

Silalahi, F. D., Putra, T. W. A., & Siswanto, E. (2022). Machine Learning Technique for Credit Card Scam Detection. *Journal of Technology Informatics and Engineering*, *1*(1), 50–79. https://doi.org/10.51903/jtie.v1i1.143

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., & Poshyvanyk, D. (2019). An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology*, *28*(4), 1–29. https://doi.org/10.1145/3345317

Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., & Filkov, V. (2015). Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, 805–816. https://doi.org/10.1145/2786805.2786860

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems*, *30*, 5998–6008. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., & Hoi, S. C. H. (2023). CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of*

the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2023), 1069–1088. https://doi.org/10.18653/v1/2023.emnlp-main.68

Weimer, W., Nguyen, T., Le Goues, C., & Forrest, S. (2009). Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, 364–374. https://doi.org/10.1109/icse.2009.5070521