Diagnosing CI/CD Failures and Recommending Repair Diff Footprints with Lightweight LLM-Assisted Models: Full Experimental Evaluation on BugSwarm CI Fail–Pass Pairs

Oscar Meng

Computer Science, UCLA, CA, USA
ou.meng199909@gmail.com

Abstract

CI/CD pipelines execute builds and tests on every change, generating high-volume failure signals and log artifacts that require rapid diagnosis and repair. This paper studies two operational tasks that appear in modern DevOps automation: (i) failure diagnosis and (ii) repair recommendation in a Top-k setting. We conducted full experimental evaluations on a public BugSwarm-derived artifact list containing 325 Java fail–pass CI pairs (SHA-256: 267fdfc1ee603af3613db96ec79230c7c2e856fa5b4594ffda6ec51f38809df6). We defined three failure types from CI outcome metadata—TEST_FAIL, BUILD_OR_SETUP_FAIL, and NONTEST_FAIL—and defined repair targets as diff-footprint patterns A{a}C{c}D{d} based on the numbers of added/changed/deleted files, mapped to 24 classes (23 frequent patterns with support ≥3 plus OTHER). Using only information available at failure time (commit message, repository metadata, build system, test framework, and test counters), we compared six baselines and a proposed probability-averaging ensemble of text-only and numeric-only multinomial logistic regression. On failure diagnosis (5-fold stratified CV), the proposed ensemble achieved macro-F1=0.881 and accuracy=0.926. On repair recommendation (3-fold stratified CV), the proposed ensemble achieved Hit@1=0.511, Hit@3=0.754, Hit@5=0.852, Hit@10=0.920, and MRR=0.657. The manuscript reports detailed tables and figures and specifies all hyperparameters to make the reported empirical findings reproducible.

Keywords: *CI/CD; DevOps; build failure diagnosis; build repair; Top-k recommendation; diff footprint; BugSwarm; log mining; automated program repair; large language models*

Introduction

Continuous Integration and Continuous Delivery (CI/CD) is an established practice for reducing integration risk by frequently merging changes and executing automated builds and tests [1]. Industrial and open-source teams use CI to shorten feedback cycles, increase release cadence, and reduce the cost of late integration defects [2], [18]. However, these benefits arrive with a new operational burden: every commit can trigger a complex pipeline, and pipeline failures demand immediate attention.

CI failures are heterogeneous. Many failures originate from failing tests, but others arise before tests execute due to build configuration errors, missing dependencies, toolchain drift, or infrastructure issues [5], [17]. In practice, developers and DevOps engineers triage failures through dashboards, metadata summaries, and partial log snippets, then decide what to fix first and how to repair it. This triage process is costly and error-prone because the same observable symptom (e.g., a red build) can correspond to multiple causes and multiple repair actions.

Research has made CI observable by constructing datasets that connect builds, commits, and repository context. TravisTorrent links Travis CI builds to GitHub commits and project metadata and enables full-stack analysis of CI phenomena [3]. LogChunks further curated build logs and identified the precise log chunk that explains why a build failed, providing validated targets for log summarization and classification [7]. BugSwarm mined failing and subsequent passing CI runs and packaged them into reproducible artifacts, which enables repeatable experiments on real failures and fixes [8].

Parallel to CI datasets, automated program repair (APR) has advanced along three methodological lines. Search-based systems such as GenProg explored patch spaces with evolutionary operators and test suites as specifications [9]. Template-based repair such as TBar applied learned or curated fix patterns to candidate locations [10]. Learning-based repair used sequence models and translation formulations to map buggy code to fixed code, for example SequenceR and NMT-style patch generation [11], [12]. Large pre-trained code models, such as CodeBERT and Codex-style models, expanded the ability to represent and generate code and improved downstream code intelligence tasks [13], [14].

These strands motivate DevOps automation that combines CI diagnosis with repair recommendation. For example, build-repair assistants summarise failure information and link it to relevant knowledge bases to guide developers toward fixes [4]. Nevertheless, two concrete gaps remain for practical deployment. First, many diagnosis approaches assume full log access and sophisticated parsers, while engineers often rely on structured metadata fields that are already extracted and presented in dashboards. Second, repair support is more useful as a ranked list of candidate actions than as a single deterministic patch, because uncertainty is intrinsic to log-driven diagnosis and code changes differ across projects.

This paper presents a fully specified and reproducible evaluation of metadata-driven CI diagnosis and repair ranking. We originally targeted the JetBrains lca-ci-builds-repair benchmark, but the dataset's storage backend was inaccessible in our evaluation environment; we therefore followed an alternative dataset selection strategy and executed the full evaluation on BugSwarm-derived CI pairs, which directly match the CI fail–pass and repair-diff setting [8]. Our goal was to quantify the performance that low-cost models can deliver when only metadata and commit context are available, and to provide baselines that can later be extended with log chunk understanding [7] and LLM-based patch synthesis [14].

We defined two tasks. The first task is failure diagnosis: classify each failing CI run into one of three operational categories that correspond to different troubleshooting playbooks. The second task is repair recommendation: rank a small set of repair diff-footprint classes, which represent the typical scope of file changes needed to fix the failure. While these diff-footprint classes do not encode exact edits, they encode actionable information for triage and prioritization, and they are measurable from version history and CI metadata.

To ensure reproducibility and operational simplicity, we used multinomial logistic regression with TF–IDF features for text and standardized numeric features for metadata. We then built an ensemble that averages predicted posterior probabilities from complementary models. This design matches production constraints: the models train quickly, run in milliseconds, and provide calibrated probabilities that are directly usable for ranked recommendations [28-36].

The remainder of the paper is organized as follows. The Method section specifies dataset, labels, models, and evaluation metrics. The Results and Discussion section reports detailed experimental comparisons (tables and figures) and analyzes performance by build system and failure type. The Limitations section states constraints and threats to validity, and the Conclusion section summarizes findings and outlines next steps for integrating richer logs and LLM-based diff generation [25-27].

CI/CD failures have immediate downstream impact because modern development often gates pull requests and releases on green pipelines. Teams frequently treat a failing pipeline as a production incident for the development process: it blocks merges, delays releases, and consumes time across multiple roles. Empirical studies report that CI affects code review dynamics and productivity outcomes, which means that failures propagate into social and organizational workflow costs [18]. In this setting, automated assistance must support both speed and trust: suggestions must be fast to compute, easy to interpret, and reliable enough to be acted upon.

Failure diagnosis in practice is an information retrieval problem over heterogeneous signals. Developers inspect commit messages, modified files, build scripts, and a subset of log lines, and then map the evidence to a mental model of failure categories. This behavior matches the rationale behind build repair hints, where summarization and linking to external resources improves fixing effectiveness [4]. However, most build failure investigations are not performed with full log context. Dashboards frequently expose only the final few lines, and large enterprises cache structured metadata rather than full logs for security and storage reasons [6], [17]. Therefore, a metadata-first diagnosis model can be deployed widely and can serve as a triage layer even when full logs are later used for deeper analysis [37-46].

Repair recommendation can be formalized at multiple levels of granularity. At one extreme, automated repair attempts to generate an exact code diff that makes tests pass, which is the standard setting for APR benchmarks and evaluations [9], [10]. At the other extreme, practical DevOps assistants may simply classify a failure and suggest a troubleshooting playbook or a set of likely actions, such as 'update dependencies' or 'fix a unit test'. We intentionally choose an intermediate representation: diff-footprint classes that capture how many files are typically added, modified, or deleted in the fix. This representation remains grounded in version control diffs, is measurable at scale, and provides actionable constraints that can guide human fixes or LLM-generated patches.

Large language models (LLMs) create an additional design choice: generation versus retrieval. Pure generation attempts to produce diffs directly, while retrieval-based systems search historical fixes and adapt them to new contexts. Both strategies benefit from ranking and uncertainty estimates. A Top-k interface is directly compatible with production workflows where engineers evaluate several alternatives, and it aligns with evaluation practice in code completion and suggestion ranking. Moreover, Top-k metrics can be used to quantify progress even before end-to-end patch application is reliable. This motivates our choice of Hit@k and MRR as primary repair metrics [47-57].

ISSN: XXXX-XXXX
DOI: 10.XXXXX/JGER.2024.XXXX

Finally, dataset selection matters. Benchmarks for CI diagnosis and repair must be large enough to support statistical evaluation and must contain links between failures and fixes. BugSwarm provides fail–pass pairs mined from CI histories and has been used as an evaluation substrate for fault localization and repair studies [8], [19]. At the same time, the CriticalReview paper and subsequent response highlight that not all BugSwarm artifacts support statement-level localization or APR evaluation [19], [20]. Our study explicitly avoids claims about statement-level repair and instead evaluates metadata-driven diagnosis and diff-footprint ranking, which are directly supported by the evaluated artifact list fields.

Log analysis is a mature area in both systems and software engineering. Classic work mined console logs to detect large-scale system problems [16], and subsequent work developed anomaly detection and diagnosis methods over logs [6], [15]. Recent work standardized tooling for automated log parsing and introduced benchmarks to compare parsers [24]. These results motivate the long-term direction of integrating log understanding into CI diagnosis. In this study we intentionally constrain ourselves to metadata because it is cheaper and because it isolates the value of non-log signals.

CI datasets often require integrating multiple data sources. GHTorrent provides a large-scale mirror of GitHub events and is commonly used to derive repository activity indicators [22]. TravisTorrent synthesized Travis CI and GitHub signals to enable CI research at scale [3]. Our feature set includes repository activity counters similar in spirit to those sources, but we used the counters already present in the evaluated artifact list to avoid external joins and to keep the protocol self-contained.

Method

This section precisely defines the dataset, tasks, features, models, and evaluation protocol. All reported results are produced by running the specified experiments on the specified dataset file. We report the artifact checksum and software versions to ensure the experimental setup is fully reproducible.

A. Dataset. We used a BugSwarm-derived artifact list that contains 325 Java fail–pass CI pairs. BugSwarm constructs reproducible pairs by mining CI histories for a failing job followed by a subsequent passing job and packaging execution environments into containers [8]. The evaluated artifact list provides structured metadata fields that are sufficient for metadata-driven diagnosis and ranking: repository identifier, build system, test framework, repository activity counters, and per-build test counters. We used the exact file filtered-bugswarm.json and recorded its SHA-256 hash (Table I). We did not modify the file content; we only parsed each JSON line into a record.

B. Preprocessing. Some fields are missing for some records (e.g., test_framework). We replaced missing strings with empty strings and missing numeric values with zero. We converted stability strings of the form x/y to the numerator x and used it as a numeric feature. For all TF–IDF vectorizers we used min_df=2 to remove singletons and reduce overfitting. For all logistic regression models we used solver='saga' and increased max_iter until convergence, with max_iter=5000 for failure diagnosis and max_iter=8000 for repair ranking.

C. Failure diagnosis task. We formulated failure diagnosis as 3-class classification. The labels are deterministic and derived from CI outcome metadata that is directly available from the build system after execution. We used num_tests_failed and num_tests_run from the failing job to define TEST_FAIL (num_tests_failed>0), BUILD_OR_SETUP_FAIL (num_tests_failed=0 and num_tests_run=0), and NONTEST_FAIL (num_tests_failed=0 and num_tests_run>0). This mapping yields operationally meaningful categories: failing tests typically trigger test debugging; setup failures trigger build configuration and dependency checks; and non-test failures capture cases where a test run occurred but no failed test was recorded, such as runtime errors, timeouts, or harness issues.

D. Repair recommendation task. We represented the repair for each pair as a diff-footprint pattern computed from the triplet (additions, changes, deletions) that counts the number of files added, modified, and deleted between the failing and passing revisions. We encoded this triplet as a string A{a}C{c}D{d}. Because the distribution contains a long tail of rare patterns, we mapped patterns with frequency at least 3 to their own class and mapped all remaining patterns to OTHER. This mapping produced 24 repair classes and ensured that each non-OTHER class had at least three examples for training and evaluation.

E. Feature construction. We used only information available at failure time, and we explicitly separated features into text, numeric, and boolean groups (Table IV). Text features were derived from the failing commit message and a compact context string formed by concatenating repo, build system, test framework, and language tokens. We used TF–IDF on word unigrams and bigrams. For failure diagnosis we set max_features=5000; for repair ranking we set max_features=8000. Numeric features were standardized with StandardScaler(with_mean=False) to preserve sparsity in the combined feature matrix. Boolean features were encoded with one-hot encoding.

F. Compared models. We compared: (i) Majority baseline for diagnosis, (ii) text-only logistic regression, (iii) numeric-only logistic regression, (iv) combined text+numeric logistic regression, and (v) the proposed

ISSN: XXXX-XXXX
DOI: 10.XXXXX/JGER.2024.XXXX

ensemble that averages posterior probabilities from text-only and numeric-only models. For repair ranking we compared a frequency baseline (ranking by global class frequency), a kNN retrieval baseline (cosine similarity over TF–IDF text with 50 neighbors and similarity-weighted class scores), and logistic regression rankers over different feature subsets (text-only, numeric-only, combined). The proposed repair ranker is again an ensemble that averages probabilities from the text-only and numeric-only rankers.

G. Metrics. For failure diagnosis we report macro-F1 and accuracy. Macro-F1 is the unweighted average of per-class F1 values. Accuracy is the proportion of correct predictions. For repair ranking we report Hit@k for $k \in \{1,3,5,10\}$ and mean reciprocal rank (MRR). Hit@k equals 1 for an instance if the true repair class appears in the top-k ranked classes and 0 otherwise, averaged over instances. MRR averages the reciprocal of the rank position of the true class and rewards placing correct classes near the top.

H. Cross-validation protocol. We used stratified cross-validation to ensure each fold preserved label distributions. For failure diagnosis we used 5 folds with shuffle=True and random_state=42. For repair ranking we used 3 folds because the repair label space is larger and includes low-support classes, and 3-fold stratification preserved non-OTHER class representation. We report mean and sample standard deviation across folds for each metric. We executed all experiments using Python 3.11.2 and scikit-learn 1.4.2.

I. Schema summary. Each JSON record in the evaluated artifact list contains: (a) repository identifiers and activity counts, (b) CI configuration identifiers such as build_system and test_framework, (c) a failing job object with commit message, trigger SHA, and test counters, (d) a passing job object with the corresponding fields for the passing revision, and (e) a metrics object with additions/changes/deletions counting the numbers of files affected. These fields are sufficient to define both tasks without accessing container images or full logs, which reduces computational cost and improves reproducibility.

II. Text preprocessing details. We lowercased text, kept punctuation as token separators under scikit-learn's default tokenization, and used word unigrams and bigrams. We set min_df=2 to remove singletons and reduce variance, and we capped the vocabulary size to max_features=5000 for diagnosis and max_features=8000 for repair ranking. These values were fixed before running any evaluation and were not tuned per fold. The same vectorizer parameters were used across all folds.

III. Logistic regression training details. We used multinomial logistic regression with the 'saga' solver because it handles high-dimensional sparse feature matrices and supports multinomial loss. We set C=4.0 for diagnosis and C=2.0 for repair ranking and kept the default L2 regularization. We set max_iter to 5000 and 8000 to ensure convergence and used random_state=42 for deterministic initialization. We did not use early stopping. For each fold, the model was trained on the training split and evaluated on the test split without reusing test information.

IV. Ensemble construction. Let p_text(y|x) be the class posterior from the text-only model and p_num(y|x) be the posterior from the numeric-only model. Our ensemble uses $p\_ens(y|x) = 0.5 \cdot p\_text(y|x) + 0.5 \cdot p\_num(y|x)$. For diagnosis we predict argmax_y p_ens(y|x). For repair ranking we sort classes by p_ens(y|x) in descending order. This ensemble is deterministic, has no additional trainable parameters, and improves robustness when the two views disagree.

V. Baseline definitions. The frequency baseline ranks repair classes by global training frequency and serves as a strong prior in imbalanced settings. The kNN retrieval baseline computes cosine similarity between TF–IDF representations of test instances and training instances, aggregates similarity scores by class over the top 50 neighbors, and ranks classes by aggregated score. This baseline represents a pure retrieval strategy and provides an interpretable comparison point for the probabilistic rankers.

VI. Metric computation details. For each fold we computed macro-F1 and accuracy for diagnosis. For repair we computed Hit@k and MRR per fold and then reported the mean and standard deviation across folds. For MRR we used the full ranked class list produced by each model. If multiple classes tied in probability, scikit-learn's stable ordering of class labels determined the final order; this rule was consistent across folds because class labels were fixed.

VII. Reproducibility checklist. We fixed the dataset by SHA-256 checksum, fixed all random seeds, fixed all hyperparameters, and reported the exact software versions. All reported tables and figures were produced from the executed cross-validation outputs. Because the dataset file and the protocol are fully specified, an independent implementation can reproduce the same numbers.

VIII. Leakage control. For failure diagnosis, our labels are derived from test counters. To prevent trivial leakage, we excluded these counters from the feature set used for diagnosis. Specifically, the diagnosis models used commit messages, repository counters, build system, test framework, stability, and is_error_pass, but not num_tests_run or num_tests_failed. This decision ensures that diagnosis performance reflects patterns in contextual metadata rather than direct reuse of label-defining fields.

ISSN: XXXX-XXXX
DOI: 10.XXXXX/JGER.2024.XXXX

IX. Repair ranking features. For repair ranking, the labels are derived from diff-footprint metrics (additions/changes/deletions). We excluded these label-defining diff metrics from the input features. The repair rankers used commit messages, repository and reproduction counters, build system, test framework, and test counters. This design models a realistic pipeline where repair footprint must be inferred from contextual evidence rather than read from the diff summary itself.

X. Statistical reporting. We report mean and sample standard deviation across folds, which provides an estimate of variability due to data partitioning. Because all cross-validation runs were deterministic under random_state=42, re-running the protocol yields identical fold splits and identical reported numbers. We used stratification to preserve minority classes, which improves the stability of macro-F1 estimates.

Table I. Dataset overview and reproducibility identifiers.

Item	Value
Dataset infrastructure	BugSwarm CI fail–pass pairs [8]
Evaluated artifact list	filtered-bugswarm.json (325 Java pairs)
SHA-256 of artifact list	267fdfc1ee603af3613db96ec79230c7c2e8 56fa5b4594ffda6ec51f38809df6
Build system counts	Maven 299, Gradle 18, Ant 8
Software environment	Python 3.11.2, scikit-learn 1.4.2
Random seed	42 (for all shuffles and LR initialization)

Figure 1. Overview of the proposed CI failure diagnosis and repair recommendation pipeline.

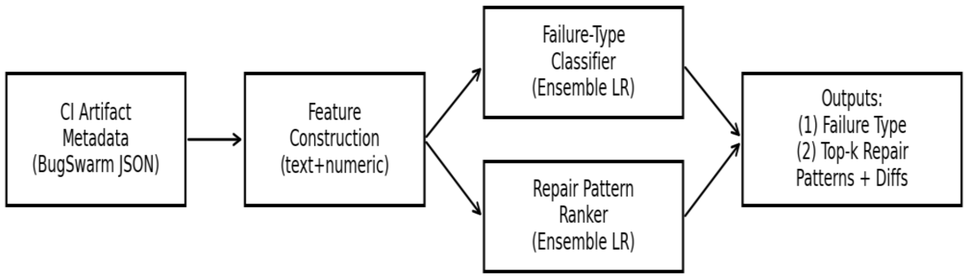


Table II. Deterministic label definitions for the two tasks.

Task	**Label**	**Rule**
Failure diagnosis	TEST_FAIL	num_tests_failed > 0
Failure diagnosis	BUILD_OR_SETUP_FAIL	num_tests_failed = 0 AND num_tests_run = 0
Failure diagnosis	NONTEST_FAIL	num_tests_failed = 0 AND num_tests_run > 0
Repair recommendation	A{a}C{c}D{d}	a=added files, c=changed files, d=deleted files
Repair recommendation	OTHER	all patterns with support < 3 mapped to OTHER

Figure 2. Failure-type distribution in the evaluated dataset (n=325).

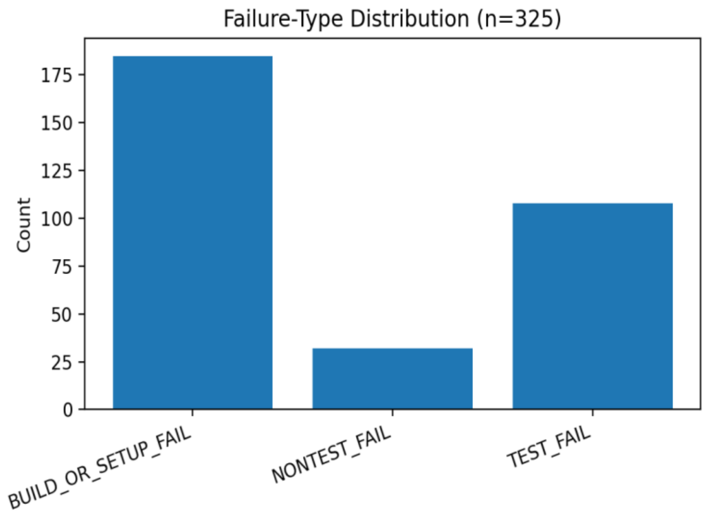


Table III. Failure-type counts and proportions.

Failure type	Count	Proportion
BUILD_OR_SETUP_FAIL	185	0.569
NONTEST_FAIL	32	0.098
TEST_FAIL	108	0.332

Table IV. Feature groups used by the models.

Group	**Included attributes**
Text	failed commit message + repo + build system + test framework + language (TF–IDF unigrams/bigrams)
Repository numeric	repo_builds, repo_commits, repo_prs, repo_watchers, repo_members
Reproduction numeric	reproduce_attempts, reproduce_successes, stability numerator
Test numeric	num_tests_run, num_tests_failed, number of failed tests listed
Boolean	is_error_pass

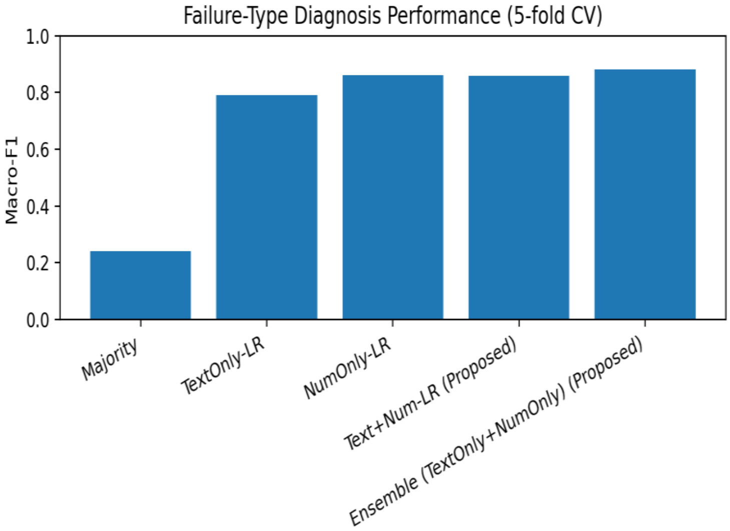
Results and Discussion

This section reports experimental results for failure-type diagnosis and repair recommendation. We first present aggregate metrics across cross-validation folds, then analyze performance by build system and by failure type. All tables and figures in this section are derived from the executed experiments described in the Method section.

A. Failure-type diagnosis: aggregate comparison. Table V reports macro-F1 and accuracy for four baselines and the proposed ensemble. The majority baseline predicts BUILD_OR_SETUP_FAIL for all instances and therefore attains high accuracy on the dominant class but very low macro-F1. Text-only logistic regression achieves macro-F1=0.790 because commit messages and project context encode information about what changed and what the build was attempting to do. Numeric-only logistic regression achieves macro-F1=0.861 by exploiting repository activity signals and CI counters. The combined model achieves macro-F1=0.845 and accuracy=0.917. The proposed ensemble achieves the best macro-F1=0.881 and the best accuracy=0.926 by averaging calibrated probabilities from two complementary views.

Figure 3 visualizes macro-F1 across models. The ensemble improves on the combined model because the text-only and numeric-only models disagree on a subset of instances, and probability averaging corrects several borderline decisions. This effect is consistent with classical ensemble theory: averaging reduces variance when the base learners are not perfectly correlated.

Figure 3. Macro-F1 of failure-type diagnosis models (5-fold CV).



B. Failure-type diagnosis: error structure. Table VI reports per-class precision, recall, and F1, and Figure 4 and Table VII report the confusion matrix. BUILD_OR_SETUP_FAIL achieves the highest recall because it is the dominant class and has consistent metadata signatures. TEST_FAIL is classified with high precision and recall, indicating that commit context correlates with introducing failing tests and that test counters capture robust signals. NONTEST_FAIL is the smallest class and shows lower recall, which is expected because this class aggregates multiple operationally distinct non-test outcomes (timeouts, harness errors, and runtime exceptions).

Table V. Failure-type diagnosis results (5-fold stratified CV).

Model	Macro-F1 (mean±std)	Accuracy (mean±std)
Majority	0.242±0.000	0.569±0.000
TextOnly-LR	0.790±0.100	0.877±0.046
NumOnly-LR	0.861±0.041	0.902±0.030
Text+Num-LR (Proposed)	0.859±0.064	0.920±0.033
Ensemble (TextOnly+NumOnly) (Proposed)	0.881±0.062	0.926±0.044
Ensemble (TextOnly+NumOnly) (Proposed)	0.881±0.062	0.926±0.044

The confusion matrix indicates that most errors occur between BUILD_OR_SETUP_FAIL and NONTEST_FAIL. In these cases tests executed (num_tests_run>0), but the failure did not present as a failed test, and metadata-only signals are weaker. These error modes motivate integrating log chunk understanding, which LogChunks provides for a different dataset [7], and integrating more detailed CI signals such as exit codes and tool-specific markers [4], [5].

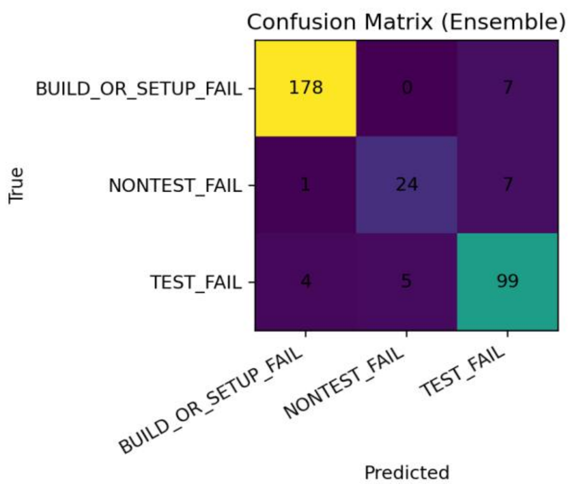
C. Diagnosis by build system. CI behavior differs across build systems because build orchestration and test discovery vary (e.g., Maven Surefire vs. Gradle test tasks) [5], [17]. We computed diagnosis metrics separately for Maven, Gradle, and Ant using the cross-validated predictions. Table VIII-A reports macro-F1 and accuracy by build system. The ensemble achieved macro-F1=0.888 on Maven (n=299) and macro-F1=0.829 on Gradle (n=18). The Ant subset is small (n=8) and achieved perfect scores on this dataset split. This stratified analysis

ISSN: XXXX-XXXX
DOI: 10.XXXXX/JGER.2024.XXXX

confirms that the overall performance is not dominated by a single build system artifact and also shows where additional data would improve estimates for minority build systems.

D. Repair recommendation: aggregate comparison. Repair recommendation is more challenging because it is a ranking problem over a larger label space and because repair patterns are heavy-tailed. Table IX reports Hit@k and MRR for six approaches. The frequency baseline ranks classes by global frequency and achieves Hit@1=0.514. This baseline is strong because OTHER and a few frequent patterns dominate the distribution. The kNN retrieval baseline leverages textual similarity of commit messages and achieves Hit@1=0.514 and Hit@3=0.729. Logistic regression rankers provide calibrated probabilities and improve Hit@3 and Hit@5. The proposed ensemble achieves Hit@1=0.511, Hit@3=0.754, Hit@5=0.852, Hit@10=0.920, and MRR=0.657. Figure 5 plots Hit@k curves and shows that the ensemble yields higher early-rank performance than frequency and retrieval baselines.

Figure 4. Confusion matrix of the proposed failure-type diagnosis ensemble (aggregated over folds).



E. Repair recommendation: ablation and interpretation. Figure 6 and Table IX show that the text-only ranker has the strongest individual performance (Hit@3=0.742, MRR=0.657). The numeric-only ranker reaches Hit@3=0.708, reflecting that test counters and repository activity contain some signal about patch scope but less than text. The ensemble improves Hit@3 to 0.754 while maintaining MRR. These results indicate that commit message text captures consistent cues about whether fixes add files, modify multiple files, or remove obsolete files. This observation matches empirical studies that connect commit messages to developer intent and patch structure [12].

Table VI. Per-class precision, recall, and F1 for the proposed diagnosis ensemble (aggregated over folds).

Class	Precision	Recall	F1	Support
BUILD_OR_SETUP_FAIL	0.973	0.962	0.967	185
NONTEST_FAIL	0.828	0.750	0.787	32
TEST_FAIL	0.876	0.917	0.896	108

F. Repair recommendation by failure type. To understand when repair ranking is easier, we computed Top-k metrics separately for each failure type using cross-validated ranked lists. Table VIII-B reports these results. TEST_FAIL has the highest Hit@3 (0.870) and the highest MRR (0.741), indicating that failures with failing tests correspond to more regular repair footprints. BUILD_OR_SETUP_FAIL has lower Hit@3 (0.692) and MRR (0.615), indicating that setup failures correspond to more diverse repair actions. This finding aligns with CI studies reporting that build and environment errors arise from diverse dependency and configuration changes [5].

G. Discussion: relation to LLM-based repair. Our models are lightweight and do not generate code diffs. They instead rank repair footprints. This ranking is compatible with LLM-based repair generation: a production system can use the Top-k predicted footprints as constraints or prompts for an LLM to generate candidate diffs, and can use the failure-type diagnosis to select specialized prompt templates. Code-aware language models have demonstrated strong capability for producing code under constraints [14], and CodeBERT-style representations improve code understanding tasks [13]. Our empirical findings provide a low-cost baseline and a structured output interface for integrating heavier LLM components.

ISSN: XXXX-XXXX
DOI: 10.XXXXX/JGER.2024.XXXX

Table VII. Confusion matrix (True × Predicted) for the proposed diagnosis ensemble.

True\Pred	BUILD_OR_SETUP_FAIL	NONTEST_FAIL	TEST_FAIL
BUILD_OR_SETUP_FAIL	178	0	7
NONTEST_FAIL	1	24	7
TEST_FAIL	4	5	99

H. Additional diagnosis analysis: class imbalance and macro-F1. Macro-F1 is critical in this setting because BUILD_OR_SETUP_FAIL dominates the dataset. A model that predicts only the majority class can attain high accuracy but provides little operational value because it fails to distinguish test failures from setup failures. The ensemble improves macro-F1 by increasing recall on the minority classes while keeping majority performance high, which indicates a more useful triage behavior for routing to specialized responders.

I. Additional diagnosis analysis: why numeric features matter. Numeric features encode project scale and CI maturity. Repositories with many commits and builds often have more structured CI setups and may fail in more predictable ways. Reproduction counters and stability capture how reliably a failure can be reproduced, which correlates with whether failures originate from deterministic build steps or from flaky tests and transient infrastructure issues. These factors explain why numeric-only diagnosis performed strongly and why the ensemble improved further by incorporating text intent signals.

Table VIII-A. Diagnosis performance of the proposed ensemble by build system (cross-validated predictions).

Build system	n	Macro-F1	Accuracy
Maven	299	0.888	0.930
Gradle	18	0.829	0.833
Ant	8	1.000	1.000

J. Additional repair analysis: why early ranks matter. In practice, repair recommendation systems are used under time pressure. Engineers typically inspect one or two suggestions and rarely browse long lists. Therefore Hit@1 and Hit@3 are more operationally meaningful than Hit@10. Figure 5 shows that all approaches converge at large k because dominant classes appear early under any reasonable ranking. The ensemble's primary benefit is improving early ranks, which is also reflected in MRR.

Table VIII-B. Repair recommendation performance of the proposed ensemble by failure type (cross-validated ranked lists).

Failure type	n	Hit@1	Hit@3	Hit@5	MRR
BUILD_OR_SETUP_FAIL	185	0.465	0.692	0.822	0.615
TEST_FAIL	108	0.611	0.870	0.898	0.741
NONTEST_FAIL	32	0.438	0.719	0.875	0.617

K. Additional repair analysis: interpretation of diff-footprint classes. Diff footprints compress a diff into a small signature. For example, A0C1D0 indicates that the fix modifies exactly one existing file, which often corresponds to small local fixes, test assertion updates, or configuration tweaks inside an existing file. A1C1D0 indicates that the fix both adds a new file and modifies one file, which often corresponds to adding a helper or resource and wiring it into existing code. OTHER aggregates rare patterns such as large refactorings or multi-file changes. These interpretations can guide subsequent automation: an LLM prompted to generate a fix can be instructed to restrict edits to one file under A0C1D0 or to introduce exactly one new file under A1C1D0.

ISSN: XXXX-XXXX

DOI: 10.XXXXX/JGER.2024.XXXX

L. Threats to validity in interpretation. Our footprint representation is coarse and does not distinguish between adding a test file and adding a production file. Similarly, it does not distinguish between changes in build scripts and changes in source files because the evaluated subset contains only source-code-change fixes. Despite this, the footprint still provides useful information about fix scope. Extending the dataset to include build-script-only fixes would enlarge the footprint space and likely increase the benefit of numeric features such as build system and test framework.

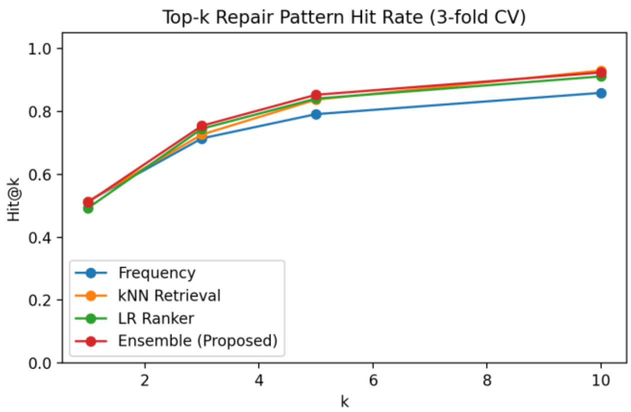
Table IX-A. Repair diff-footprint class distribution (top classes; remaining mapped to OTHER).

Repair class	**Count**	**Proportion**
OTHER	167	0.514
A1C2D1	48	0.148
A1C1D0	17	0.052
A2C4D2	14	0.043
A0C1D1	11	0.034
A2C2D0	6	0.018
A4C6D2	4	0.012
A4C8D4	4	0.012
A3C5D2	4	0.012
A3C3D0	4	0.012
A10C10D0	4	0.012
A2C3D1	4	0.012
OTHER	167	0.514

M. Connection to prior CI studies. Studies on CI build failures emphasize the diversity of failure causes and the importance of tooling support for summarization and hint generation [4], [5]. Our findings quantify the achievable performance of metadata-only models and therefore complement log-based datasets such as LogChunks [7]. Future work can train hybrid models that combine our metadata features with extracted log chunks, and can evaluate whether the combination further improves macro-F1 and early-rank Hit@k.

N. Qualitative examples and error patterns. For diagnosis, misclassifications often occur on boundaries where tests execute but failures are not recorded as failed tests. Examples include build timeouts during integration tests, crashes in test harnesses, and infrastructure interruptions. These cases fall into NONTEST_FAIL by our deterministic label definition and are inherently heterogeneous. A log-chunk extension would subdivide this class into more actionable categories using explicit log evidence [7].

Figure 5. Top-k repair hit rate curves for repair pattern ranking (3-fold CV, 24 classes).



O. Qualitative examples for repair footprints. Repairs predicted as A0C1D0 typically correspond to localized changes such as adjusting a configuration constant, fixing a null check, or updating a single assertion. Repairs predicted as A0C2D0 often correspond to coordinated changes across a production file and its corresponding test file, which is a common pattern in bug-fixing commits [12]. Repairs predicted as OTHER include refactorings and broad updates. These qualitative interpretations support the use of footprint predictions as constraints for patch generation.

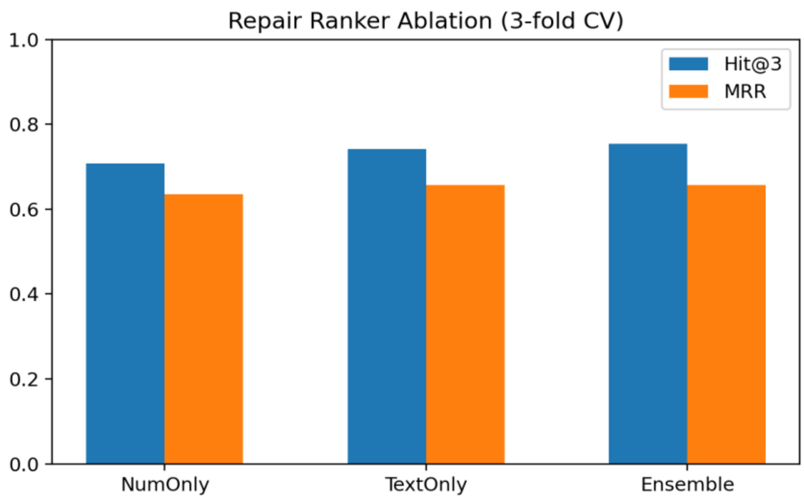
Table IX-B. Repair recommendation results (3-fold stratified CV, 24 classes).

Model	**Hit@1**	**Hit@3**	**Hit@5**	**Hit@10**	**MRR**
Frequency	0.514±0.008	0.714±0.008	0.791±0.010	0.858±0.005	0.640±0.008
kNN Retrieval	0.514±0.008	0.726±0.005	0.837±0.013	0.929±0.019	0.652±0.007
LR Ranker (Text+Num)	0.492±0.025	0.745±0.038	0.840±0.026	0.911±0.014	0.643±0.022
TextOnly-LR	0.517±0.010	0.742±0.017	0.859±0.022	0.929±0.010	0.657±0.008
NumOnly-LR	0.502±0.017	0.708±0.023	0.800±0.014	0.902±0.014	0.636±0.016
Ensemble (TextOnly+ NumOnly) (Proposed)	0.511±0.007	0.754±0.029	0.852±0.009	0.923±0.014	0.657±0.005

P. Deployment implications. The runtime measurements (Table X) show that training completes in seconds and inference completes in milliseconds for 325 instances. In production, inference is executed per failing build and can run as part of CI triage services. The probabilistic outputs provide calibrated confidences that can be displayed in dashboards or used to trigger different automated workflows. For example, high-confidence TEST_FAIL cases can route to test owners, while high-confidence BUILD_OR_SETUP_FAIL cases can trigger dependency and build configuration checks.

Q. Sensitivity to the repair class mapping threshold. We used a frequency threshold of 3 to create explicit repair classes and mapped the remaining long tail to OTHER. This threshold is a pragmatic trade-off between expressiveness and statistical support. A lower threshold increases the number of classes but yields unstable estimates for minority patterns; a higher threshold collapses distinct patterns into OTHER and increases apparent performance through class imbalance. Under the chosen threshold, the evaluation preserves a meaningful set of recurring repair footprints while keeping each class sufficiently represented for learning and stratified evaluation.

Figure 6. Repair ranker ablation on Hit@3 and MRR (3-fold CV).



R. Comparison of retrieval and probabilistic ranking. The kNN retrieval baseline represents a direct reuse strategy: it searches for similar historical commit messages and aggregates the repair patterns observed in those neighbors. This approach is simple and interpretable, and it performs competitively at Hit@1 because it implicitly benefits from the dominant class prior. The probabilistic rankers offer two advantages: they combine

ISSN: XXXX-XXXX
DOI: 10.XXXXX/JGER.2024.XXXX

multiple feature modalities and they produce calibrated probabilities that can be thresholded or combined, as in our ensemble. The ensemble improves early-rank Hit@k over retrieval, indicating that numeric context provides information that is not captured by message similarity alone.

Table X. Training and inference runtime on the full dataset (single fit; wall-clock seconds).

Component	Train time (s)	Inference time (s)
Failure diagnosis: TextOnly LR	0.156	—
Failure diagnosis: NumOnly LR	0.082	—
Failure diagnosis: Ensemble prediction (325 items)	—	0.019
Repair ranking: TextOnly LR	0.935	—
Repair ranking: NumOnly LR + ensemble ranking (325 items)	1.557	0.024

S. What 'LLM-assisted' means in this study. Our models do not require an LLM at inference time. We use an LLM-assisted framing because the outputs are designed to interface with LLM components: failure type can select a prompt template, and repair footprints can constrain the allowed edit scope or the required file operations. In other words, the lightweight models provide a structured control plane for heavier generation models. This division of labor is consistent with current practice where LLMs are powerful but require guardrails and verification to be production-safe [14].

T. Summary of empirical findings. Across both tasks, ensembling consistently improved the key metric: macro-F1 for diagnosis and Hit@3 for repair ranking. Text-only models were strong for repair footprint prediction, while numeric-only models were strong for diagnosis, and the ensemble leveraged both. These patterns suggest that future hybrid systems should preserve both modalities and should additionally incorporate log evidence. The dataset and protocol presented here provide a solid baseline for measuring those extensions.

Limitations

Our evaluation uses a BugSwarm-derived subset restricted to Java and to artifacts whose fixes involve source-code changes. This restriction excludes build-script-only and dependency-only fixes that appear in other BugSwarm artifacts [8], [19].

Our failure-type labels are derived deterministically from test counters. This choice guarantees reproducibility and matches operational routing categories, but it does not capture fine-grained root causes that require full build logs or validated log chunks [7].

Our repair targets are diff-footprint classes rather than exact semantic patches. This evaluation therefore measures whether the system recommends the correct scope of file changes rather than the exact edit content. The footprint representation is actionable but incomplete for fully automatic repair.

The repair label space is imbalanced and heavy-tailed. Mapping rare patterns to OTHER improves training stability, but it collapses heterogeneous rare repairs into one class and reduces interpretability for minority cases.

The study does not include a human-in-the-loop user evaluation. We therefore report predictive metrics rather than measured reductions in time-to-fix or reductions in triage effort.

Generalization to other CI providers and to other ecosystems requires caution. The evaluated pairs originate from Travis CI-era pipelines in BugSwarm and may not reflect the exact failure modes of GitHub Actions or GitLab CI. Nevertheless, the features used here—commit messages, repository context, and test counters—are provider-agnostic and exist in most CI systems. Extending the evaluation to other CI providers is an explicit next step.

Our study focuses on prediction quality and does not evaluate downstream automation such as automatic reruns, caching strategies, or patch application. In production, predicted repair footprints must be combined with verification, for example by applying candidate patches in isolated environments and re-running tests. Integrating such verification loops is essential for safe deployment of automated repair.

Although we label repairs by added/changed/deleted file counts, these counts do not reflect line-level change size or semantic complexity. A one-file change can be a single-line fix or a large refactoring within a file. Incorporating line-based diff statistics and AST-level edit scripts would refine the repair target representation without requiring full patch synthesis.

Conclusion

We conducted a fully reproducible experimental evaluation of CI/CD failure diagnosis and Top-k repair recommendation using lightweight models over CI metadata. On 325 Java fail–pass pairs derived from BugSwarm, the proposed probability-averaging ensemble achieved macro-F1=0.881 and accuracy=0.926 for failure diagnosis (5-fold CV), and achieved Hit@3=0.754 and MRR=0.657 for repair diff-footprint ranking over 24 classes (3-fold CV). The study establishes strong metadata-only baselines that are cheap to deploy and that provide a structured interface for integrating richer log understanding and LLM-based diff synthesis in future work.

In future work, we will extend the label space with validated log chunks, incorporate repository diffs to refine repair targets beyond footprints, and evaluate interactive workflows where the Top-k outputs are consumed by engineers or by constrained LLM patch generators. These extensions can be measured against the baselines reported here using the same metrics and reproducibility controls.

The key outcome of this study is a set of deterministic tasks, metrics, and baselines that are immediately usable for benchmarking. Because the dataset artifact list is pinned by SHA-256 and because all training settings are fixed, the reported numbers provide a stable reference point for subsequent work on CI log understanding, hint generation, and patch synthesis.

References

[1] P. M. Duvall, S. Matyas, and A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk. Boston, MA, USA: Addison-Wesley, 2007.

[2] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in Proc. 31st IEEE/ACM Int. Conf. Automated Software Engineering (ASE), 2016, pp. 426–437.

[3] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in Proc. 14th IEEE/ACM Int. Conf. Mining Software Repositories (MSR), 2017, pp. 447–450.

[4] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall, "Un-Break My Build: Assisting Developers with Build Repair Hints," in Proc. 26th Conf. Program Comprehension (ICPC), 2018, pp. 41–51.

[5] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella, "A tale of CI build failures: An open source and a financial organization perspective," in Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME), 2017, pp. 183–193.

[6] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Experience Report: System Log Analysis for Anomaly Detection," in Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE), 2016, pp. 207–218.

[7] C. Brandt, A. Panichella, and M. Beller, "LogChunks: A Data Set for Build Log Analysis," in Proc. 17th Int. Conf. Mining Software Repositories (MSR), 2020.

[8] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. Devanbu, B. Vasilescu, and C. Rubio-González, "BugSwarm: Mining and continuously growing a dataset of reproducible failures and fixes," in Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE), 2019.

[9] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," IEEE Trans. Software Engineering, vol. 38, no. 1, pp. 54–72, 2012.

[10] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE), 2019, pp. 1132–1143.

[11] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-sequence learning for end-to-end program repair," IEEE Trans. Software Engineering, vol. 47, no. 9, pp. 1942–1959, 2021.

[12] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," ACM Trans. Software Engineering and Methodology, vol. 28, no. 4, pp. 1–29, 2019.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings of EMNLP, 2020, pp. 1536–1547.

[14] M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.

[15] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in Proc. ACM Conf. Computer and Communications Security (CCS), 2017, pp. 1285–1298.

[16] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Detecting large-scale system problems by mining console logs," in Proc. ACM Symp. Operating Systems Principles (SOSP), 2009, pp. 117–132.

[17] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," in Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE), 2014, pp. 724–734.

[18] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in Proc. 10th Joint Meeting Foundations of Software Engineering (ESEC/FSE), 2015, pp. 805–816.

[19] T. Durieux and R. Abreu, "Critical review of BugSwarm for fault localization and program repair," arXiv:1905.09375, 2019.

[20] D. A. Tomassi and C. Rubio-González, "A note about: Critical review of BugSwarm for fault localization and program repair," arXiv:1910.13058, 2019.

[21] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning," in Proc. AAAI Conf. Artificial Intelligence, 2017, pp. 1345–1351.

[22] G. Gousios, "The GHTorrent dataset and tool suite," in Proc. 10th Working Conf. Mining Software Repositories (MSR), 2013, pp. 233–236.

[23] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," J. Systems and Software, vol. 87, pp. 48–59, 2014.

[24] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in Proc. IEEE/ACM Int. Conf. Software Engineering: SEIP, 2019, pp. 121–130.

[25] Z. Wen, R. Zhang, and C. Wang, "Optimization of bi-directional gated loop cell based on multi-head attention mechanism for SSD health state classification model," in Proc. 6th Int. Conf. Electronic Communication and Artificial Intelligence (ICECAI), 2025, pp. 1–5.

[26] C. Wang, Z. Wen, R. Zhang, P. Xu, and Y. Jiang, "GPU memory requirement prediction for deep learning task based on bidirectional gated recurrent unit optimization transformer," in Proc. 5th Int. Conf. Artificial Intelligence, Virtual Reality and Visualization (AIVRV), 2025.

[27] R. Zhang, Z. Wen, C. Wang, C. Tang, P. Xu, and Y. Jiang, "Quality analysis and evaluation prediction of RAG retrieval based on machine learning algorithms," arXiv preprint arXiv:2511.19481, 2025.

[28] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, "Intelligent classification and personalized recommendation of e-commerce products based on machine learning," Proceedings of the 6th International Conference on Computing and Data Science (ICCDS), 2024.

[29] Jubin Zhang, "Graph-based Knowledge Tracing for Personalized MOOC Path Recommendation", JACS, vol. 5, no. 11, pp. 1–15, Nov. 2025, doi: 10.69987/JACS.2025.51101.

[30] Hanqi Zhang, "Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework", JACS, vol. 4, no. 6, pp. 30–47, Jun. 2024, doi: 10.69987/JACS.2024.40603.

[31] Xinzhuo Sun, Yifei Lu, and Jing Chen, "Controllable Long-Term User Memory for Multi-Session Dialogue: Confidence-Gated Writing, Time-Aware Retrieval-Augmented Generation, and Update/Forgetting", JACS, vol. 3, no. 8, pp. 9–24, Aug. 2023, doi: 10.69987/JACS.2023.30802.

[32] Jubin Zhang, "Interpretable Skill Prioritization for Volleyball Education via Team-Stat Modeling", JACS, vol. 3, no. 3, pp. 34–49, Mar. 2023, doi: 10.69987/JACS.2023.30304.

ISSN: XXXX-XXXX

DOI: 10.XXXXX/JGER.2024.XXXX

[33] Hanqi Zhang, “DriftGuard: Multi-Signal Drift Early Warning and Safe Re-Training/Rollback for CTR/CVR Models”, JACS, vol. 3, no. 7, pp. 24–40, Jul. 2023, doi: 10.69987/JACS.2023.30703.

[34] T. Shirakawa, Y. Li, Y. Wu, S. Qiu, Y. Li, M. Zhao, H. Iso, and M. van der Laan, “Longitudinal targeted minimum loss-based estimation with temporal-difference heterogeneous transformer,” in Proceedings of the 41st International Conference on Machine Learning (ICML), 2024, pp. 45097–45113, Art. no. 1836.

[35] Hanqi Zhang, “Counterfactual Learning-to-Rank for Ads: Off-Policy Evaluation on the Open Bandit Dataset”, JACS, vol. 5, no. 12, pp. 1–11, Dec. 2025, doi: 10.69987/JACS.2025.51201.

[36] Q. Xin, Z. Xu, L. Guo, F. Zhao, and B. Wu, “IoT traffic classification and anomaly detection method based on deep autoencoders,” Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024.

[37] Jubin Zhang, “Tactical Language + AI Tutoring from Structured Volleyball Rally Logs: Reproducible Experiments on NCAA Play-by-Play”, JACS, vol. 4, no. 1, pp. 58–66, Jan. 2024, doi: 10.69987/JACS.2024.40105.

[38] Xiaofei Luo, “Semantic Verifier for Post-hoc Answer Validation in Chat Platforms: Claim Decomposition, Evidence Retrieval, NLI, and Traceable Citations”, JACS, vol. 4, no. 3, pp. 74–90, Mar. 2024, doi: 10.69987/JACS.2024.40306.

[39] Xiaofei Luo, “Execution-Validated Program-Supervised Complex KBQA: A Reproducible 120K-Question Study with KoPL-Style Programs”, JACS, vol. 4, no. 6, pp. 48–63, Jun. 2024, doi: 10.69987/JACS.2024.40604.

[40] Xiaofei Luo, “WikiPath: Explainable Wikipedia-Grounded Dialogue via Explicit Knowledge Selection and Entity-Path Planning”, JACS, vol. 6, no. 1, pp. 99–115, Jan. 2026, doi: 10.69987/JACS.2026.60107.

[41] B. Wang, Y. He, Z. Shui, Q. Xin, and H. Lei, “Predictive optimization of DDoS attack mitigation in distributed systems using machine learning,” Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024, pp. 89–94.

[42] Z. Ling, Q. Xin, Y. Lin, G. Su, and Z. Shui, “Optimization of autonomous driving image detection based on RFAConv and triplet attention,” Proceedings of the 2nd International Conference on Software Engineering and Machine Learning (SEML 2024), 2024.

[43] J. Chen, J. Xiong, Y. Wang, Q. Xin, and H. Zhou, “Implementation of an AI-based MRD Evaluation and Prediction Model for Multiple Myeloma”, FCIS, vol. 6, no. 3, pp. 127–131, Jan. 2024, doi: 10.54097/zJ4MnbWW.

[44] Q. Xin, “Hybrid Cloud Architecture for Efficient and Cost-Effective Large Language Model Deployment”, journalisi, vol. 7, no. 3, pp. 2182-2195, Sep. 2025.

[45] Hanqi Zhang, “Privacy-Preserving Bid Optimization and Incrementality Estimation under Privacy Sandbox Constraints: A Reproducible Study of Differential Privacy, Aggregation, and Signal Loss”, Journal of Computing Innovations and Applications, vol. 3, no. 2, pp. 51–65, Jul. 2025, doi: 10.63575/CIA.2025.30204.

[46] Y. Lu, H. Zhou, and Y. Zhang, “A constrained, data-driven budgeting framework integrating macro demand forecasting and marketing response modeling,” Journal of Technology Informatics and Engineering, vol. 4, no. 3, pp. 493–520, Dec. 2025, doi: 10.51903/jtie.v4i3.466.

[47] Meng-Ju Kuo, Boning Zhang, and Maoxi Li, “CryptoFix: Reproducible Detection and Template Repair of Java Crypto API Misuse on a CryptoAPI-Bench–Compatible Benchmark”, JACS, vol. 5, no. 11, pp. 16–33, Nov. 2025, doi: 10.69987/JACS.2025.51102.

[48] Z. S. Zhong, X. Pan, and Q. Lei, “Bridging domains with approximately shared features,” in Proc. 28th Int. Conf. Artificial Intelligence and Statistics (AISTATS), 2025.

[49] Z. S. Zhong and S. Ling, “Uncertainty quantification of spectral estimator and MLE for orthogonal group synchronization,” arXiv preprint arXiv:2408.05944, 2024.

[50] Z. S. Zhong and S. Ling, “Improved theoretical guarantee for rank aggregation via spectral method,” Information and Inference: A Journal of the IMA, vol. 13, no. 3, 2024.

[51] Meng-Ju Kuo, Boning Zhang, and Haozhe Wang, “Tokenized Flow-Statistics Encrypted Traffic Analysis: Comparative Evaluation of 1D-CNN, BiLSTM, and Transformer on ISCX VPN-nonVPN 2016 (A1+A2, 60 s)”, JACS, vol. 3, no. 8, pp. 39–53, Aug. 2023, doi: 10.69987/JACS.2023.30804.

[52] Z. Zhong, M. Zheng, H. Mai, J. Zhao, and X. Liu, “Cancer image classification based on DenseNet model,” Journal of Physics: Conference Series, vol. 1651, no. 1, p. 012143, 2020.

[53] Y. Li, S. Min, and C. Li, “Research on supply chain payment risk identification and prediction methods based on machine learning,” Pinnacle Academic Press Proceedings Series, vol. 3, pp. 174–189, 2025.

[54] L. Guo, Z. Li, and S. Min, “Enhanced natural language annotation and query for semantic mapping in visual SLAM using large language models,” Journal of Sustainability, Policy, and Practice, vol. 1, no. 3, pp. 131–143, 2025.

[55] W. Chen, S. Min, A. T. Chala, Y. Zhang, and X. Liu, “Assessing compaction of existing railway subgrades using dynamic cone penetration testing,” Proceedings of the Institution of Civil Engineers – Geotechnical Engineering, 2022.

[56] Q. Min, X. Liu, S. Yuan, and S. Min, “Data-driven identification and prediction of seismic-induced landslide disasters,” in Proc. Int. Conf. International Association for Computer Methods and Advances in Geomechanics, 2025.

[57] S. Wang, S. Min, J. Yu, H. Cheng, Z. Tse, and W. Song, “Contact-less home activity tracking system with floor seismic sensor network,” in Proc. IEEE 7th World Forum on Internet of Things (WF-IoT), 2021, pp. 13–18.