HiLL PUBLISHING

Engineering Advances, 2026, 6(3), 228-232
https://www.hillpublisher.com/journals/ea/
ISSN Online: 2768-7961

Research on Automatic Analysis of Software Testing Results and Defect Feedback Mechanisms Based on Intelligent Agents

Yue Cheng

Heinz College of Information Systems and Public Policy, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

How to cite this paper: Yue Cheng. (2026) Research on Automatic Analysis of Software Testing Results and Defect Feedback Mechanisms Based on Intelligent Agents. *Engineering Advances*, 6(3), 228-232.
DOI: 10.26855/ea.2026.09.016

Received: July 26, 2026
Accepted: August 22, 2026
Published: September 17, 2026

***Corresponding author:** Yue Cheng, Heinz College of Information Systems and Public Policy, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

© 2026 by the author(s).
This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives (CC BY-NC-ND) license, which permits non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited and is not modified or adapted.
https://creativecommons.org/licenses/by-nc-nd/4.0/

Abstract

Continuous integration and automated testing environments generate heterogeneous evidence, including execution logs, exception traces, coverage records, failed test cases, and defect tickets, while conventional scripts are generally limited to rule matching and result aggregation and often lose semantic relations among failures, causes, and subsequent handling steps. An intelligent-agent workflow is developed around result perception, semantic judgment, defect localization, feedback scheduling, and retesting control, where failure logs, execution traces, exception messages, and historical defect records are converted into structured evidence for agent reasoning, and confidence gating together with feedback priority is used to prevent low-confidence diagnoses from entering repair actions directly. The resulting process connects test execution, defect identification, responsibility assignment, repair candidate generation, and regression verification in a continuous processing chain, reducing repeated manual inspection of large failure sets while preserving traceable decision evidence; the mechanism also provides a practical basis for automated defect handling, cross-project adaptation, and continuous testing in complex software systems where test outputs are frequent, heterogeneous, and difficult to interpret with static rules alone.

Keywords

Intelligent agents; software testing; test result analysis; defect feedback; automated repair

Introduction

Continuous integration platforms accumulate large volumes of test outputs, and a failed case may simultaneously involve log fragments, exception stacks, dependency states, and version differences, making a return code or fixed rule insufficient for locating the fault and estimating its downstream impact. Recent large-language-model applications in software testing already cover test generation, program repair, test execution, and fault analysis, and a review of 102 studies shows that semantic models are increasingly used as analytical and decision-support components within testing activities [1]. Intelligent agents add task state, tool invocation, and feedback memory to this setting, allowing evidence identification, defect judgment, repair suggestion, and regression verification to proceed as a connected sequence and reducing the manual effort required to inspect repeated test failures.

1. Intelligent-Agent-Assisted Architecture for Software Test Result Perception and Decision-Making

1.1 Multi-Agent Role Allocation and Task Routing

Failed test cases, exception stacks, log fragments, and runtime states differ in data structure and diagnostic value,

so a coordinator agent separates incoming evidence according to task identifiers, code versions, execution environments, and exception types before assigning log interpretation, fault diagnosis, and defect triage to specialized agents. Test-case identifiers, dependency states, and contextual indexes remain attached to each task, allowing agents to exchange results around the same execution instance while reducing inconsistent interpretations caused by repeated parsing. Research on multi-agent software engineering has treated role allocation, task decomposition, and inter-agent communication as central design elements of agent-based systems [2]; with state records and routing rules in place, the coordinator can redirect tasks according to the quality of preceding outputs and send incomplete or conflicting results to supplementary analysis, preserving a continuous and traceable task chain for subsequent defect localization.

1.2 Test Evidence Representation and Confidence Gating

Once testing outputs enter the analysis stage, the evidence-processing module normalizes log events, exception types, assertion results, execution traces, and environment states, then links them to test cases, code commits, and execution timestamps so that the analysis agent can distinguish code faults from dependency failures, environment mismatches, and test-script errors. As shown in Fig. 1, the collected evidence flows into the agent orchestration layer, where the analysis, diagnosis, and triage agents process the information in sequence before the results enter confidence scoring and gating for downstream routing. Each diagnostic output records the predicted defect category, evidence location, reasoning basis, and confidence score; agent-based test-execution research has incorporated feedback checking and output constraints into the execution process and reported low false-positive and false-negative rates under the evaluated settings [3], allowing high-confidence cases to proceed to defect localization, medium-confidence cases to request additional traces or logs, and low-confidence cases to enter manual review.

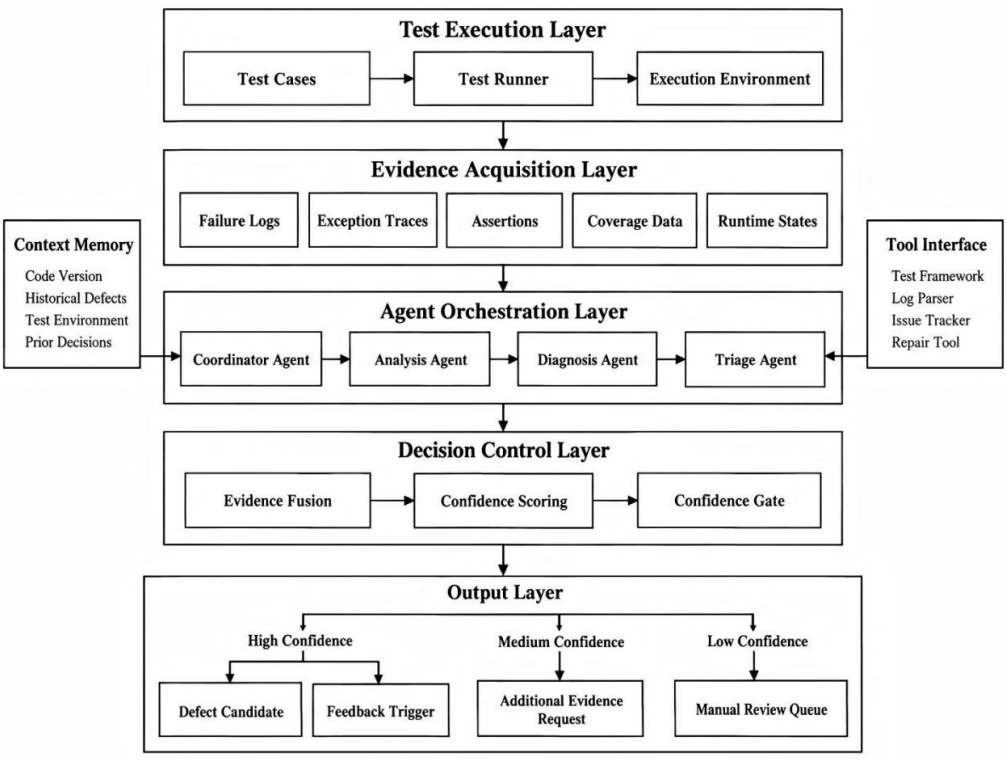

Figure 1. Intelligent-agent-based test result perception and decision architecture.

2. Automatic Interpretation of Software Testing Results and Defect Feedback Control Mechanism

2.1 Failure Semantic Parsing and Defect Localization

Raw test outputs are organized as an evidence set $E = \{L, S, A, C, D\}$, where L denotes execution logs, S

exception stacks, A assertion results, C coverage information, and D recent code changes; the analysis agent aligns these records by test-case identifier, commit version, timestamp, and execution environment, then extracts exception classes, stack frames, failed assertions, covered functions, and modified code regions into a unified representation. Candidate files or functions are ranked with $Score(f) = w_1S_s + w_2S_a + w_3S_c + w_4S_d$, where S_s, S_a, S_c, and S_d describe stack relevance, assertion relevance, coverage relevance, and change relevance, while the weights are determined according to the reliability of each evidence source; locations below the decision threshold are withheld from downstream defect handling to reduce weak associations produced by isolated log messages. MuTAP combines large-language-model-based test generation with mutation testing and reports a mutation score of 93.57% on synthetic buggy code while detecting up to 28% more faulty human-written snippets under its evaluated setting, supporting the use of mutation outcomes as supplementary evidence when ranking suspicious code locations [4].

2.2 Feedback Triggering and Repair Candidate Generation

After suspicious locations have been ranked, the diagnosis agent packages the result into a structured feedback object $F = \{loc, type, evidence, conf, context\}$, containing the suspected location, defect category, supporting evidence, confidence value, and execution context; a feedback score $F_s = \alpha C_l + \beta C_e + \gamma C_r$ combines localization confidence C_l, cross-source evidence consistency C_e, and failure reproducibility C_r, and only cases exceeding the configured threshold are sent to the repair agent. The repair agent reads the target function, failed assertion, neighboring code, exception context, and relevant historical defect record, then generates a restricted set of candidate patches for conditions such as boundary errors, null access, state inconsistencies, and parameter mismatches, with every candidate retaining the modified location, rationale, affected tests, and expected behavioral change. A repair framework evaluated on 1,974 defective Alloy specifications used a Repair Agent together with an Instructor Agent and returned analyzer errors or counterexamples to subsequent repair rounds, providing a practical reference for converting diagnostic output into controlled, traceable repair feedback [5].

2.3 Iterative Retesting and Feedback Convergence Control

Each candidate patch is executed in an isolated test environment where the previously failed cases are rerun before the affected module enters regression testing, and the controller records the remaining failed-test count N_f, newly introduced failures N_n, coverage variation $\triangle C$, and diagnostic difference between consecutive rounds $\triangle D$. A candidate is accepted when the original failure disappears, $N_n = 0$, and the affected regression set remains stable; if new failures arise or the evidence pattern changes materially, the current execution trace is returned to the diagnosis stage and a revised candidate is generated from the updated context. To prevent repetitive repair loops, the controller sets a maximum iteration count K_{max}, a minimum diagnostic-change threshold $\mathcal{E}$, and a consecutive-failure counter r; processing stops and enters manual review when K_{max} is reached, when $\triangle D < \mathcal{E}$ persists across successive rounds, or when repeated candidates fail to alter the observed fault state, keeping automated feedback within a defined computational and diagnostic boundary.

3. Defect Feedback Performance Validation and Engineering Applicability Analysis

3.1 Sample Selection and Evaluation Metric Setting

Validation data are organized into program repair, defect assignment, and fault-prediction tasks while retaining the original project boundaries. Automated program repair uses 2,672 bugs from 12 projects and applies Exact Match to compare generated patches with reference fixes; bug triage uses 11 open-source projects, where issue descriptions, code snippets, developer labels, and issue types are stored as separate fields rather than merged across incompatible label spaces [6-7].

Metrics remain task-specific during processing: Exact Match is retained for repair, developer candidates and issue categories are recorded for triage, and AUC together with rule violations and process metrics is used for risk ranking. The ASP.NET Core dataset contains 1,339 issue reports collected from January to June 2020, averaging 223 reports per month across 84 developers and 197 issue types; these dimensions are retained directly when configuring the feedback queue.

3.2 Comparison of Automated Repair and Defect Assignment Results

Repair results are calculated at project level before differences in Exact Match are compared across adaptation

settings. In the journal version of the domain-adaptation experiment, TFix records an average gain of 13.05% and CodeXGLUE 48.78%, while synthetic-data-supported zero-shot adaptation gives average gains of 5.76% and 17.62%, respectively; on the triage side, developer recommendation and issue-type prediction are retained as separate outputs, and only ranked candidates with sufficient confidence enter the assignment queue.

Table 1 consolidates the published sample sizes and numerical outcomes used for comparison. The triage evaluation fields follow the performance dimensions summarized from 123 eligible studies [8], while the fault-prediction dataset contains 29 Java projects, 58,125 commits, 33,865 faults, approximately 1.8 million SonarQube rule violations, and 48 product/process metrics [9]; values from different publications remain separated during interpretation.

Table 1. Published experimental data related to defect analysis and feedback

Validation task	Sample size	Evaluation field	Published result	Processing stage
Domain-adapted APR	2,672 bugs/12 projects	Exact Match	TFix +13.05%; CodeXGLUE +48.78%	Repair generation
Zero-shot APR	Unlabeled target projects	Exact Match	TFix +5.76%; CodeXGLUE +17.62%	Project adaptation
Bug triage	11 open-source projects	Developer/issue type	1,339 issues/6 months	Feedback routing
Triage workload	84 developers/197 issue types	Queue load	223 issues/month	Assignment control
Fault prediction	29 Java projects	AUC	>95% for selected features	Risk ranking
Static evidence	58,125 commits/33,865 faults	48 metrics	About 1.8M rule violations	Evidence scoring

3.3 Cross-Project Transfer and Deployment Boundaries

Before cross-project deployment, the target repository is mapped against the source project in programming language, log schema, defect labels, and commit structure; the repair model then loads historical repair pairs associated with matching defect classes, while projects without labeled fixes use synthetic bug-fix pairs for lightweight adaptation. Major dependency upgrades or changes to the testing framework trigger a new sample extraction and confidence-distribution check, and a held-out set is used to verify repair outputs before the updated configuration enters the feedback queue.

For risk ranking, SonarQube rules, code size, modification frequency, and historical fault records are converted into a common feature table, after which the agent recalculates feature distributions for the target repository and determines alert thresholds on validation data. Selected feature sets in the published Java-project dataset reached AUC values above 95%, which provides a reference benchmark for deployment; a different programming language or static-analysis tool requires a new feature calibration rather than direct reuse of the original threshold.

4. Conclusion

Intelligent agents allow software testing outputs to be organized as continuous evidence derived from logs, exception stacks, assertions, coverage information, and code changes, while specialized agents handle interpretation, diagnosis, assignment, and repair. Confidence gating restricts low-confidence decisions from entering repair operations, and structured feedback preserves defect locations, supporting evidence, and execution context for subsequent retesting. Published project data provide measurable references for domain adaptation, defect triage, and static-analysis-based fault assessment. In practical deployment, programming languages, log schemas, defect labels, and tool versions require project-specific calibration, while manual review remains necessary for uncertain or repeatedly unresolved cases.

References

[1] Wang J, Huang Y, Chen C, et al. Software testing with large language models: survey, landscape, and vision. IEEE Trans Softw Eng. 2024;50(4):911-936.

[2] He J, Treude C, Lo D. LLM-based multi-agent systems for software engineering: literature review, vision, and the road ahead. ACM Trans Softw Eng Methodol. 2025;34(5):124.

[3] Salva S. Reliable execution of natural language test cases for GUI applications using LLM agents. Softw Qual J. 2026;34:33.

[4] Moradi Dakhel A, Nikanjam A, Majdinasab V, et al. Effective test generation using pre-trained large language models and mutation testing. Inf Softw Technol. 2024;171:107468.

[5] Alhanahnah M, Rashedul Hasan M, Xu L, et al. An empirical evaluation of pre-trained large language models for repairing declarative formal specifications. Empir Softw Eng. 2025;30:149.

[6] Zirak A, Hemmati H. Improving automated program repair with domain adaptation. ACM Trans Softw Eng Methodol. 2024; 33(3):65.

[7] Aung TWW, Wan Y, Huo H, et al. Multi-triage: a multi-task learning framework for bug triage. J Syst Softw. 2022;184:111133.

[8] Nagwani NK, Suri JS. An artificial intelligence framework on software bug triaging, technological evolution, and future challenges: a review. Int J Inf Manag Data Insights. 2023;3(1):100153.

[9] Lomio F, Moreschini S, Lenarduzzi V. A machine and deep learning analysis among SonarQube rules, product, and process metrics for fault prediction. Empir Softw Eng. 2022;27:189.