2026 IEEE 50th Annual Computers, Software, and Applications Conference (COMPSAC) | 979-8-3315-4497-3/26/$31.00 ©2026 IEEE | DOI: 10.1109/COMPSAC69091.2026.00147

Artifact-Constrained Agentic Testing for Black-Box System Testing under Actuarial and Regulatory Constraints

Hi Kuen Yu[1], Jacky Keung[1], Man On Wong[2], Yicheng Sun[1,*], Rachel Samantha Chandra[1], Yihan Liao[1] and Xiaoxue Ma[3]
[1]Department of Computer Science, City University of Hong Kong, Hong Kong, China
[2]University of Economics Ho Chi Minh City, Vietnam
[3]Department of Electronic Engineering and Computer Science, Hong Kong Metropolitan University, Hong Kong, China
hikuenyu2-c@my.cityu.edu.hk, jacky.keung@cityu.edu.hk, wongmanon@ueh.edu.vn, yicsun2-c@my.cityu.edu.hk, rchandra3-c@my.cityu.edu.hk, yihanliao4-c@my.cityu.edu.hk, kxma@hkmu.edu.hk
*Corresponding author

***Abstract*—System-level testing of insurance software systems is challenging due to long-lived legacy architectures, distributed actuarial logic, regulatory-driven conditional behavior, and exception-heavy workflows, where correctness is defined by compliance with evolving domain constraints rather than deterministic outputs. Existing approaches either focus on artifact-centric validation, rely heavily on human-driven execution and diagnosis, or generate test artifacts in a one-shot manner without sustained adaptation during execution, limiting robustness and reproducibility at the system level. This paper proposes *Artifact-Constrained Agentic Testing* (ACAT), a multi-agent framework that structures black-box system testing as a closed-loop process of planning, execution, diagnosis, and repair, in which LLM-based agents operate under explicitly bounded capabilities and interact with the system under test exclusively through tool-mediated execution. We evaluate ACAT through a controlled industrial study on an insurance broker management system using 50 real-world use cases. The results show that artifact-constrained agentic testing significantly improves test executability and execution stability compared to human-driven testing, while exposing a complementary subset of system-level failures. These findings suggest that agent-assisted testing can enhance system-level automation in complex, regulated software systems without replacing human expertise.**

***Index Terms*—Black-Box System Testing, Multi-Agent Frameworks, Automated End-to-End Testing, Large Language Model, Insurance Technology**

I. Introduction

Insurance-related software systems pose persistent challenges for system-level testing because they combine long-lived legacy architectures [1]–[3], distributed actuarial logic, and regulatory-driven conditional behavior [4]–[6]. Unlike transactional systems with deterministic input–output relationships, correctness in insurance platforms depends on whether components satisfy actuarial rules, regulatory invariants, and business constraints [2], [4], [6]. Thus, system-level testing emphasizes constraint satisfaction and cross-component consistency rather than fixed expected outputs [5].

This difficulty is amplified by black-box legacy cores, limited interfaces, and business rules scattered across back-end services, configuration artifacts, calculation sheets, and regulatory guidelines [2]–[5]. Regulatory compliance introduces jurisdiction-specific branching behavior, while broker and claims workflows involve incomplete documentation, conflicting third-party data, fraud indicators, and other exception-heavy cases [6]–[9]. These factors create a large, evolving scenario space that is difficult to test exhaustively and reproduce reliably.

Despite advances in test automation, system-level testing in insurance remains largely human-driven. Testers still interpret requirements, select scenarios, design test data, repair brittle scripts, and judge correctness [10]–[12]. UI- and script-centric automation remains fragile under interface changes and dynamic workflows [13]–[15], and the test oracle problem persists because correctness often requires actuarial or regulatory judgment rather than deterministic comparison [14], [16], [17].

This paper investigates whether agent-assisted system-level testing can improve robustness while preserving black-box constraints. We introduce *Artifact-Constrained Agentic Testing* (ACAT), where LLM-based agents operate under bounded capabilities and interact with the system under test only through tool-mediated execution. ACAT structures testing as an artifact-driven loop of planning, execution, diagnosis, and repair, enabling adaptive system-level testing without access to internal system logic. In summary, our primary contributions can be summarized as follows:

- We formalize the actuarial oracle problem as a system-level testing challenge where correctness cannot be reduced to outputs under black-box constraints.
- We design an artifact-constrained, multi-agent testing pipeline that enables closed-loop planning, execution, and repair without assuming access to internal system logic.
- Through an industrial study on 50 real-world use cases, we empirically show that agent-assisted testing improves execution robustness but exposes a complementary subset of system-level failures compared to human testing.

2836-3795/26/$31.00 ©2026 IEEE
DOI 10.1109/COMPSAC69091.2026.00147

II. Related Work

Prior work on financial and insurance software testing has proposed specification-driven validation, claims auditing, and risk- or coverage-oriented test reduction. For example, existing approaches validate policy administration or claims logic against calculation sheets, product documents, or formalized business specifications [3], [4]. However, these approaches primarily address artifact-specific validation and do not fully support end-to-end behavioral testing across changing interfaces, workflows, and execution environments [3], [4], [6].

AI-based testing research has explored test case generation, test suite optimization, and automated test development using machine learning and LLMs [18]–[21]. In insurance-related contexts, generative techniques have also been proposed to generate functional, regression, and negative test scenarios [8]. Nevertheless, many AI-based testing methods remain one-shot or weakly coupled to execution feedback, producing artifacts that still require human validation, repair, and domain interpretation [12], [13], [17].

Agentic AI systems introduce planning, tool use, task decomposition, and feedback-driven iteration for software engineering tasks [10], [22], [23]. Early agent-based testing studies show that coordinated agents can organize generation, execution, and reporting more effectively than monolithic prompts [11]. However, existing agentic approaches have not been systematically applied to closed-loop, system-level testing in actuarially complex and regulated insurance environments [13], [17]. This paper addresses this gap by integrating planning, execution, diagnosis, and repair under explicit artifact constraints and black-box execution conditions.

III. Methodology

This study proposes Artifact-Constrained Agentic Testing (ACAT), a multi-agent framework driven by LLMs for automated system-level black-box testing of complex web applications. ACAT targets domains in which internal system logic is inaccessible and correctness is governed by domain-specific constraints, such as insurance systems. The testing process is formalized as an iterative reasoning–execution–diagnosis loop, in which autonomous agents operate under explicitly bounded capabilities and interact with the system under test (SUT) exclusively via tool-mediated execution. By structuring testing as a closed-loop, artifact-driven process, ACAT enables adaptive test generation and repair while preserving black-box assumptions, experimental control, and reproducibility.

A. System Architecture

The system architecture is organized into three logical layers: (i) an agent orchestration layer, (ii) a tool invocation and execution layer, and (iii) the SUT accessed through a real browser runtime. The agent orchestration layer is hosted within a Model Context Protocol (MCP) environment, which provides the runtime for LLM-based agents and embeds an MCP client for standardized communication. The tool invocation layer consists of MCP-compliant servers that expose browser automation and execution capabilities. The SUT is accessed exclusively through these tools via a production-grade browser. Fig. 1 shows the system architecture for MCP-based tool execution in system testing, while Fig. 2 illustrates the LLM-based multi-agent workflow implementing a closed-loop system testing pipeline.

This layered design enforces a strict separation between reasoning and execution. LLM-based agents do not directly interact with the browser or the SUT; all external interactions are mediated through MCP-exposed tools. This constraint preserves black-box testing conditions, ensures that execution behavior is observable and reproducible, and prevents uncontrolled side effects. Decision-making is confined to the agent layer, while all execution is delegated to tool servers.

B. Multi-Agent Testing Pipeline

Within the MCP host environment, the framework instantiates a multi-agent testing pipeline (Planner, Coder and Repair) PCR that decomposes black-box end-to-end testing into coordinated but explicitly bounded stages. The pipeline formalizes testing as a stateful, artifact-driven, closed-loop process, in which intermediate representations, execution artifacts, diagnostic classifications, and repair outcomes are persistently stored and reused across iterations. An LLM-based agent orchestrator manages phase transitions, maintains intermediate artifacts, and enforces execution constraints such as retry budgets and termination conditions. Although the pipeline is sequential in structure, it operates iteratively, enabling refinement of test artifacts based on execution outcomes.

C. Planning Agent

The Planning Agent transforms high-level natural-language inputs, such as functional requirements, usage scenarios, or acceptance criteria, into a structured test plan. Inputs include the SUT entry point (e.g., URL and authentication information) and execution constraints such as available test data.

The output is a structured representation specifying test objectives, ordered user interaction steps, expected outcomes, and coverage priorities. This plan serves as a stable abstraction of test intent, separating requirement interpretation from implementation details. Explicit encoding of assumptions and expectations improves traceability between requirements, generated scripts, and execution outcomes, and constrains downstream agents to a well-defined semantic scope.

D. Coding Agent

The Coding Agent translates the structured test plan into executable browser-based test scripts. Generated scripts encode navigation sequences, user interactions, assertions, and required setup or teardown logic. When the test plan specifies alternative paths or multiple scenarios, the agent may synthesize multiple scripts corresponding to these behaviors.

Script generation is constrained by two classes of artifacts: (i) the structured test plan and associated generation rules, which define permissible behaviors and implementation boundaries, and (ii) a Code Book containing previously validated implementation-level patterns. The Code Book is

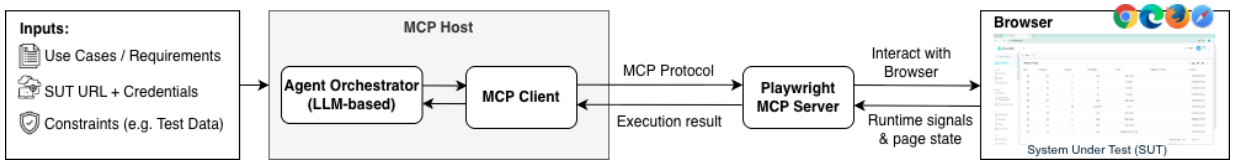

Fig. 1: System Architecture: MCP-based Tool Execution for System Testing.

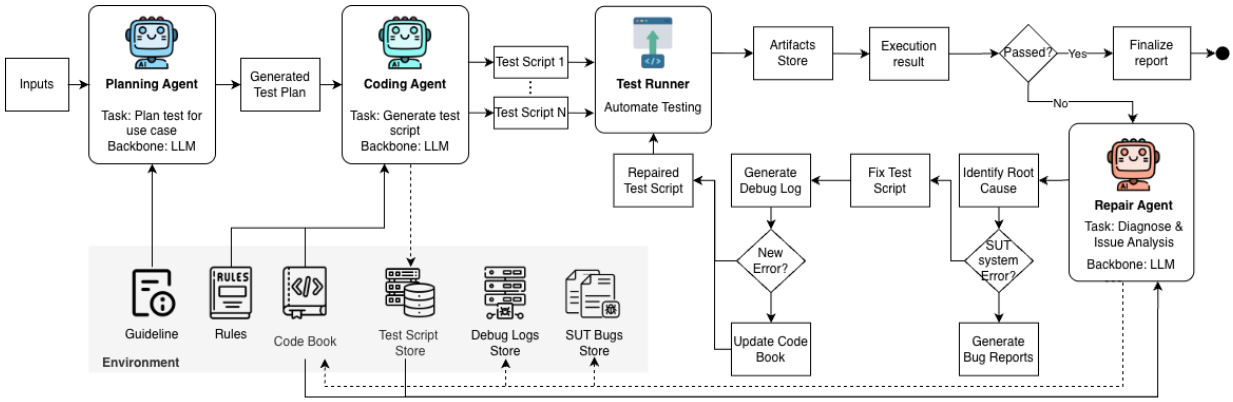

Fig. 2: Multi-Agent Workflow: Closed-loop System Testing Pipeline.

consulted as a reference to guide implementation decisions and does not define test intent or control flow.

E. Test Runner

Test execution is performed by a Test Runner that invokes browser automation through a Playwright MCP server. The server executes generated scripts against the SUT using a real browser instance and returns structured execution results, including pass/fail outcomes, execution logs, traces, screenshots, and runtime page-state signals.

The Test Runner is treated as an execution service rather than an autonomous agent. This distinction reinforces the separation between reasoning and action and ensures that all execution behavior is externally observable and repeatable. Execution artifacts are persisted in an artifact store and made available for diagnostic analysis.

F. Repair Agent

When a test execution fails, a Repair Agent is invoked to perform diagnostic analysis and determine appropriate follow-up actions. The Repair Agent operates in two stages: root-cause analysis and conditional outcome handling. Fig 3 shows the workflow for test failure diagnosis and resolution.

Root-cause analysis The Repair Agent analyzes the failing test script together with associated execution artifacts, including logs, browser traces, DOM snapshots, screenshots, and runtime signals. Based on this evidence, failures are classified as: (i) test-level faults (e.g., unstable locators, synchronization issues, incorrect assertions); (ii) SUT-level defects, attributable to system behavior; or (iii) execution or environmental anomalies. Classification decisions are based on observable execution evidence, such as whether failures persist across retries with corrected locators or synchronization, the presence of explicit system error messages, and consistency of failure behavior across executions.

Conditional repair and reporting Repair actions are attempted only for failures classified as test-level faults. In such cases, the agent generates a revised test script addressing the identified cause and submits it to the Test Runner for re-execution. Repair attempts are bounded by predefined limits to prevent uncontrolled iteration and overfitting.

Failures classified as SUT-level defects are not repaired. Instead, the Repair Agent generates a structured SUT bug report summarizing the observed behavior and supporting evidence. These reports are stored in a dedicated SUT bug repository and excluded from further repair cycles.

Repair Agent Instruction Template:

Inputs. Failing test script; execution artifacts (logs, traces, screenshots); codebook.

Task. Analyze the failing test execution and classify the failure as either a test-level fault or a system-under-test (SUT) defect. If the failure is classified as a test-level fault, attempt to repair the test by applying validated patterns from the codebook while preserving the original test intent.

Constraints. A maximum of k repair attempts is permitted. The intended behavior of the test must not be modified. All repairs must reference validated implementation patterns from the codebook.

Output. Either `TEST_FIXED`, accompanied by a Test Fix Report, or `SYSTEM_BUG`, accompanied by a System Bug Report.

Post-condition. After task completion (either

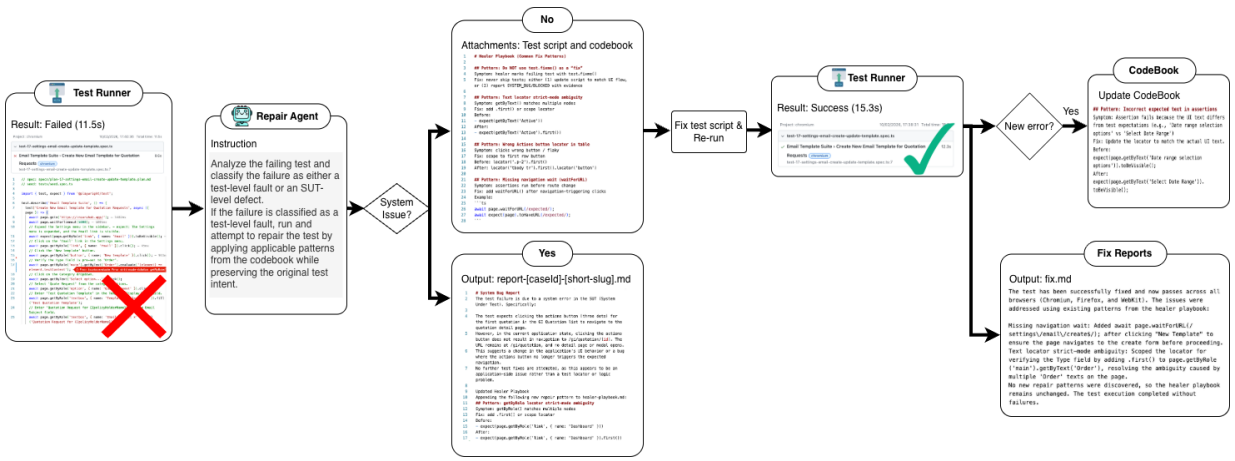

Fig. 3: Repair Agent Workflow for Test Failure Diagnosis and Resolution.

`TEST_FIXED` or `SYSTEM_BUG`), append any newly identified reusable *repair pattern* to the codebook. This step is skipped if no new pattern is identified or validated during the current run.

Codebook The codebook is a structured repository of previously validated implementation-level patterns for UI test automation (e.g., robust locator scoping, navigation prerequisites, synchronization strategies, and assertion adjustments). It does not define test intent or control flow; instead, it serves as a reference consulted by the Coding Agent and Repair Agent when generating or repairing Playwright scripts. The codebook is updated only after a repair is successfully validated through re-execution: when the Repair Agent resolves a test-level fault and the revised script passes, the agent may append a generalized pattern (symptom $\rightarrow$ fix) to the codebook to reduce recurrence of similar execution failures in subsequent cases.

Codebook Entry Example:

Symptom. Missing element when opening a quotation detail page directly.
Cause. Missing prerequisite navigation.
Fix Pattern. Navigate to the list view, assert element visibility, and then select the target record.

G. Closed-Loop Execution Control

The framework operates as an autonomous closed-loop pipeline that iteratively refines test artifacts through planning, execution, and repair cycles. Starting from an initial test plan, scripts are generated, executed, evaluated, and, when applicable, repaired and re-executed. Iteration terminates when one of the following conditions is met: (i) all generated scripts pass successfully; (ii) a predefined retry or repair budget is exceeded; or (iii) failures are classified as SUT-level defects.

Upon termination, execution results and collected artifacts are aggregated into a final test report summarizing outcomes, identified defects, and applied repairs. As all interactions with the SUT are mediated through MCP and all intermediate artifacts are preserved, the process yields an auditable and reproducible testing record.

IV. Empirical Setup

A. System Under Test and Use Cases

The SUT is an industrial insurance broker management system supporting broker-facing workflows, including customer onboarding, quotation, policy issuance, endorsement processing, financial processing, and claims-related operations. The SUT is accessed through a production-like web interface under black-box conditions. No access is provided to internal source code, database state, or proprietary business-rule implementations.

The evaluation uses 50 real-world system-level use cases derived from operational workflows and organizational test artifacts. The use cases cover nine functional categories and include 14 low-, 21 medium-, and 15 high-complexity workflows. All use cases are executed under an organizational administrator role.

B. Compared Conditions

We compare three black-box system testing conditions. In the human-driven baseline (H), eight testers with prior web application testing experience design and implement Playwright-based end-to-end tests. In ACAT, the proposed framework performs planning, script generation, execution, diagnosis, and repair without codebook support. In ACAT+, agents consult a codebook containing validated implementation-level repair patterns. Each human tester executes all 50 use cases, while ACAT and ACAT+ each execute the 50 use cases once.

C. Research Questions

The study addresses two research questions:

- **RQ1:** How does Artifact-Constrained Agentic Testing affect the robustness and reproducibility of system-level test artifacts compared to human-driven testing?
- **RQ2:** To what extent can artifact-constrained multi-agent testing expose system-level failures, and how does its defect exposure differ from that of human-driven testing?

TABLE I: Executability Rate by Use Case Complexity

Complexity	H	ACAT	ACAT+
High	0.39	1.00	0.80
Medium	0.25	0.57	0.86
Low	0.58	0.14	0.86
Overall	0.39	0.58	0.84

D. Evaluation Metrics

For each execution, we collect generated scripts, execution logs, and replay outcomes. All scripts are independently re-run in a controlled environment to assess reproducibility and observable behavior. Consistent with system testing principles articulated by Myers [24], evaluation focuses on test artifact robustness and defect-revealing capability. Metrics are based on observable execution behavior.

- **Executability Rate** measures the proportion of generated test scripts that can be successfully re-run without execution-level errors, capturing robustness and reproducibility of test artifacts.
- **Execution Failure Depth** measures how far a non-executable test progresses before termination, computed as the ratio of the stopping line to total lines of code. Median values are reported to reduce sensitivity to skewed distributions and outlier failures common in execution-level data.
- **System-Level Failure Rate** measures the proportion of executable scripts that expose observable system-level failures, serving as a proxy for defect-revealing effectiveness.

V. Results and Discussion

A. Test Artifact Robustness

Table I reports executability by use case complexity. Overall executability increases from 0.39 under human-driven testing to 0.58 under ACAT and 0.84 under ACAT+. This indicates that agentic testing improves script robustness, but the largest gain occurs when reusable implementation-level knowledge is available through the codebook.

The improvement from ACAT to ACAT+ shows that reasoning-driven automation is insufficient for robust system-level testing. Without codebook support, ACAT performs inconsistently, achieving full executability on high-complexity cases but failing frequently on low-complexity cases. This suggests that many failures are caused by recurring implementation-level issues, such as unstable locators, missing prerequisite navigation, and synchronization errors. Codebook support mitigates these problems by reusing validated repair patterns, leading to stable execution across all complexity levels.

Failure-depth analysis further supports this interpretation. For high-, medium-, and low-complexity failures, median failure depth under ACAT+ is 0.14, 0.15, and 0.45, compared with 0.44, 0.48, and 0.66 under human-driven testing. This indicates that ACAT+ localizes execution problems earlier and avoids prolonged unstable execution.

TABLE II: Exposed System-Level Failures by Category

Category	H	ACAT	ACAT+
CRUD	15	0	1
UI/frontend	6	0	6
Error handling	7	0	0
Business logic	2	0	2
Access control	4	0	0
Performance	1	0	0
Total	35	0	9

Answer to RQ1: ACAT improves the robustness and reproducibility of system-level test artifacts compared with human-driven testing. The improvement is strongest when ACAT is supported by a codebook: executability increases from 0.39 under human-driven testing to 0.84 under ACAT+, while failure-depth results indicate earlier localization of execution-level problems. These findings suggest that artifact constraints and reusable repair patterns are central to stabilizing agent-generated system tests.

B. System-Level Failure Exposure

Table II summarizes the system-level failures exposed by each condition. Human-driven testing reports 35 failures across 400 scripts, corresponding to a failure rate of 0.09. ACAT+ reports 9 failures across 50 scripts, corresponding to a higher failure rate of 0.18 among executed scripts. ACAT without codebook support reports no system-level failures, indicating that insufficient execution robustness prevents tests from reaching meaningful system states.

Across all conditions, 27 use cases exhibit system-level failures. Human-driven testing identifies failures in 21 use cases, while ACAT+ identifies failures in 9 use cases, with only 3 overlapping cases. This partial overlap suggests that agent-assisted and human-driven testing expose complementary defect profiles. Human testers reveal a broader range of failures, including error handling, access control, and observability issues. ACAT+ primarily exposes externally observable CRUD, UI/frontend, and selected business-logic failures.

Answer to RQ2: ACAT can expose system-level failures once sufficient execution robustness is achieved. ACAT without codebook support does not expose system-level failures, while ACAT+ identifies 9 failures across 50 scripts. Its defect profile differs from human-driven testing, with limited overlap in failing use cases. This indicates that codebook-supported agentic testing complements, rather than replaces, human-driven system testing.

C. Practical Implications

The results indicate that ACAT+ is most useful as a reproducible execution and repair layer for black-box system testing. Human testers remain stronger in exploratory validation and in identifying broader categories of defects, particularly those involving access control, error handling, and observability. By contrast, ACAT+ improves execution stability and reliably exposes externally observable failures once test scripts reach meaningful system states. Therefore, agent-assisted testing should be deployed as a complementary mechanism that reduces low-level automation maintenance and

allows human testers to focus on domain-specific judgment and exploratory analysis.

VI. Threats to Validity

External Validity: A potential threat to external validity arises from the scope of the empirical evaluation. The study is conducted on a single industrial insurance broker management system, using 50 real-world use cases derived from its operational workflows. Although the system exhibits characteristics common to insurance platforms—such as black-box constraints, rule-intensive logic, and exception-heavy processes—results may not directly generalize to other insurance systems or regulated domains with different architectures, user interfaces, or regulatory environments. To mitigate this threat, the evaluation focuses on system-level properties (e.g., test executability, failure exposure) that are largely independent of specific business rules, and uses authentic, production-derived use cases rather than synthetic benchmarks. Nevertheless, further studies across additional systems and domains are required to assess the generalizability of the findings.

Construct Validity: Construct validity may be affected by how system-level testing effectiveness is operationalized and measured. In this study, robustness is assessed through executability rate and execution failure depth, while effectiveness is approximated by the exposure of observable system-level failures. These metrics capture externally observable testing outcomes but do not provide a complete measure of test adequacy or defect severity, particularly for failures that require deep domain expertise to interpret. In addition, classification of failures as test-level faults or system-level defects relies on execution evidence and predefined diagnostic criteria, which may introduce classification bias. To reduce this risk, failure classification is grounded in observable execution artifacts (logs, traces, screenshots) and applied consistently across all conditions. Nonetheless, alternative metrics or complementary qualitative analysis could further strengthen construct validity.

VII. Conclusion

This paper presented ACAT, an artifact-constrained, multi-agent framework for black-box system testing in regulated insurance software. By integrating planning, execution, diagnosis, and repair through tool-mediated browser automation, ACAT improves reproducibility of end-to-end test artifacts. In an industrial study with 50 real-world use cases, codebook-supported ACAT increased executability from 0.39 to 0.84 and exposed a complementary subset of system-level failures. The results suggest agent-assisted testing is most useful as a reproducible execution and repair layer that complements human exploratory testing. Future work will examine hybrid human-agent workflows, broader domain evaluations, and long-term codebook evolution.

References

[1] J. Anderson, "Modernizing legacy insurance systems: Cloud migration strategies and microservices architecture for operational agility."

[2] N. B. Goolla, "Ai in insurance claims processing: Balancing innovation with implementation challenges," *European Journal of Computer Science and Information Technology*, vol. 13, no. 27, pp. 58–71, 2025.

[3] N. Rao, V. Krishnasubramaniam, S. Waghamale, S. Sondur, A. Ansari, S. Iyer, A. Singh, and M. Vijayalakshmi, "Data validation for eligibility criteria and death benefit claims for insurance policies," in *2023 4th International Conference for Emerging Technology (INCET)*. IEEE, 2023, pp. 1–7.

[4] A. Datar, A. Zare, A. A, R. Venkatesh, S. Kumar, and U. Shrotri, "Automated validation of insurance applications against calculation specifications," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2022, pp. 55–60.

[5] P. K. Gollapudi, "Cloud-native ai-driven test automation framework for insurance software systems. 5," 2023.

[6] P. C. Shekhar, "Testing approaches in life insurance: Accelerated and fluid less underwriting amidst data-driven dynamics," 2024.

[7] F. Elkourdi, C. Wei, L. Xiao, Z. YU, and O. Asan, "Exploring current practices and challenges of hipaa compliance in software engineering: scoping review," *IEEE Open Journal of Systems Engineering*, vol. 2, pp. 94–104, 2024.

[8] M. Santhosh, P. Rengaraju, C.-H. Lung, Y. Cao *et al.*, "Automation of medical insurance post-claims using ocr and rag models," in *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2025, pp. 2026–2031.

[9] P. C. Shekhar, "Testing for the unexpected: Ensuring insurance system stability during covid-19," 2022.

[10] D. Milchevski, G. Frank, A. Hätty, B. Wang, X. Zhou, and Z. Feng, "Multi-step generation of test specifications using large language models for system-level requirements," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, 2025, pp. 132–146.

[11] B. Sherifi, K. Slhoub, and F. Nembhard, "The potential of llms in automating software testing: From generation to reporting," *arXiv preprint arXiv:2501.00217*, 2024.

[12] N. B. Hasan, M. A. Islam, J. Y. Khan, S. Senjik, and A. Iqbal, "Automatic high-level test case generation using large language models," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 674–685.

[13] F. Ricca, A. Marchetto, and A. Stocco, "A multi-year grey literature review on ai-assisted test automation," *Information and Software Technology*, p. 107799, 2025.

[14] V. Garousi, N. Joy, Z. Jafarov, A. B. Keleş, S. Değirmenci, E. Özdemir, and R. Zarringhalami, "Ai-powered software testing tools: A systematic review and empirical assessment of their features and limitations," *arXiv preprint arXiv:2409.00411*, 2024.

[15] F. Ricca, B. García, M. Nass, and M. Harman, "Next-generation software testing: Ai-powered test automation," *IEEE Software*, vol. 42, no. 4, pp. 25–33, 2025.

[16] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[17] F. Molina, A. Gorla, and M. d'Amorim, "Test oracle automation in the era of llms," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–24, 2025.

[18] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.

[19] A. Ahammad, M. El Bajta, and M. Radgui, "Automated software testing using machine learning: A systematic mapping study," in *2024 10th International Conference on Optimization and Applications (ICOA)*. IEEE, 2024, pp. 1–6.

[20] A. Mehmood, Q. M. Ilyas, M. Ahmad, and Z. Shi, "Test suite optimization using machine learning techniques: A comprehensive study," *IEEE Access*, vol. 12, pp. 168 645–168 671, 2024.

[21] Y. Zhang, "New approaches to automated software testing based on artificial intelligence," in *2024 5th International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*. IEEE, 2024, pp. 806–810.

[22] T. Zhu, L. C. Cordeiro, and Y. Sun, "Reqinone: A large language model-based agent for software requirements specification generation," in *2025 IEEE 33rd International Requirements Engineering Conference (RE)*. IEEE, 2025, pp. 449–457.

[23] X. Fei, X. Zheng, and H. Feng, "Mcp-zero: Active tool discovery for autonomous llm agents," *arXiv preprint arXiv:2506.01056*, 2025.

[24] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.