Automated Test Generation and Code Evaluation with Custom Copilot Architecture

Galal Mohamed*, Tasneem Mohammed†, Laurance Ashraf‡, Raneem Mousa§
Zewail City, Egypt
Emails: s-galal.qassas@zewailcity.edu.eg, s-tasneem.ahmed@zewailcity.edu.eg,
s-laurence.shehata@zewailcity.edu.eg, s-raneem.adam@zewailcity.edu.eg

***Abstract*—Ensuring software correctness and reliability is a critical challenge. AI, particularly Large Language Models (LLMs), promises automated test generation to reduce manual effort and improve coverage. However, LLM-generated code and tests may contain hallucinations, compilation errors, and incorrect logic. This research presents a custom multi-agent Copilot architecture for automated test identification, implementation, evaluation, and iterative refinement with human-in-the-loop supervision. Early results demonstrate improved test coverage, reduced manual intervention, and mitigation of AI hallucination errors [1], [2].**

***Index Terms*—AI Testing, Large Language Models, Automated Test Generation, PyTest, Multi-Agent Systems, HITL, Software Quality, Test Coverage**

I. Introduction

Manual testing is labor-intensive and error-prone, and traditional automated tools struggle with complex or AI-generated code [11]. LLM-driven code generation accelerates development but introduces hallucinated functions, incorrect assertions, and potential critical failures [4]. This research proposes a custom Copilot architecture leveraging multi-agent AI systems for automated test case identification, generation, execution, and evaluation. Human-in-the-loop (HITL) supervision mitigates hallucinations, ensuring correctness and robustness [8], [9].

II. Literature Review

A. Traditional Testing Methods

Search-Based Software Testing (SBST) tools like EvoSuite and random-testing tools like Randoop form the foundation of automated testing [2]. They often produce tests that are hard to read or maintain and struggle with complex test inputs or AI-generated code [11].

B. AI/LLM-Based Testing Advances

Early AI frameworks used deep learning, reinforcement learning, and evolutionary algorithms to dynamically generate test cases based on execution patterns and historical data [4]. Multi-agent pipelines with iterative feedback and HITL supervision improve reliability of AI-generated tests [8], [9]. Comparative studies show traditional automated tools outperform LLMs in key metrics such as coverage and assertion correctness [6], [9].

Fig. 1. Multi-agent Copilot architecture for automated test identification, implementation, evaluation, and dashboarding.

III. System Architecture

The architecture consists of interacting agents with iterative evaluation and HITL supervision, as shown in Fig.1.

A. Input Code Layer

Receives raw user code and identifies functions, return behaviors, exceptions, and execution paths.

B. Test Case Identification Agent

Performs static analysis to detect core functionality, edge cases, error-handling paths, and expected outputs, producing a structured test plan.

C. Human-in-the-Loop (HITL) Supervision

Reviews test plans, corrects hallucinated cases, and ensures completeness before implementation.

D. Test Implementation Agent

Converts test plans into executable PyTest scripts, performs incremental code writing, executes tests, and adapts to evaluator feedback.

E. Evaluator Agent

Executes tests and collects metrics: pass/fail, runtime errors, coverage, and security alerts. Feedback informs iterative refinement.

F. Iteration Control

Repeats up to 15 cycles or until coverage exceeds 90%, ensuring high quality without infinite loops.

G. JSON Telemetry + Dashboard

Stores logs of code states, metrics, and errors. The dashboard visualizes coverage, execution history, and security alerts.

IV. Experimental Setup

Component	Configuration
Programming Language	Python 3.10
Testing Framework	PyTest + Coverage.py
Data Logging	JSON
Execution Model	Agent-based iterative loop

TABLE I
EXPERIMENTAL ENVIRONMENT CONFIGURATION

Python scripts of varying complexity were used as input. No external datasets were required.

V. Preliminary Results

Metric	Observation
Coverage Achieved	90–96%
Iterations Required	6–11 cycles
Human Intervention	Minimal after stabilization
Security Detection	Functional, requires expansion

TABLE II
EARLY RESULTS METRICS

Coverage gradually increased across cycles. Initial hallucinations decreased with iterative feedback, and failing tests were self-corrected.

VI. Error and Hallucination Analysis

Error Type	Description
Hallucinated Functions	Tests for non-existent code elements
Incorrect Assertions	Wrong expected output assumptions
Over-strict Conditions	Failures due to minor variations
False Security Alerts	Misclassified high-risk warnings

TABLE III
OBSERVED ERRORS AND HALLUCINATIONS

Mitigation strategies include HITL supervision, iterative evaluation, and stricter specification parsing [1], [4].

VII. Phase 3 Development Plan

- AI-Driven Bug Fixing — Auto-repair of code
- Multi-Language Support — Java/JS/C# integration
- Deep Coverage Analysis — Path- and branch-based metrics
- CI/CD Integration — Continuous deployment and monitoring
- Advanced Vulnerability Scanning — CVE mapping and static analysis

VIII. Risks, Limitations, and Mitigation

Key risks include:

- LLM hallucination causing incorrect test generation
- Slow execution on large codebases
- Coverage metrics not guaranteeing correctness
- Single-language constraints
- Working on python code only

Mitigation strategies:

- HITL supervision and iterative feedback loops
- Incremental caching and selective execution
- Semantic testing and property-based checks
- Modular multi-language support

IX. Conclusion

This study demonstrates a custom multi-agent Copilot architecture for automated test generation and code evaluation. Preliminary results show improved coverage, reduced human effort, and mitigation of AI hallucinations. Future work will extend to fully autonomous software quality assurance.

References

[1] 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. 1, pp. 911–936, 2024. doi: 10.1109/TSE.2024.3394119.

[2] G. Kathiresan, "Automated Test Case Generation with AI: A Novel Framework for Improving Software Quality and Coverage," *World Journal of Advanced Research and Reviews*, vol. 23, no. 2, pp. 2880–2889, 2024. doi: 10.30574/wjarr.2024.23.2.2463.

[3] A. Fan, M. Lyubarskiy, B. Gokkaya, S. Sengupta, M. Harman, S. Yoo, and J. M. Zhang, "Large Language Models for Software Engineering: Survey and Open Problems," in *Proc. 2023 IEEE/ACM Int. Conf. Softw. Eng.: Future of Softw. Eng. (ICSE-FoSE)*, Melbourne, Australia, 2023, pp. 31–53. doi: 10.1109/ICSE-FoSE59111.2023.00013.

[4] A. Celik and Q. H. Mahmoud, "A Review of Large Language Models for Automated Test Case Generation," *Machine Learning and Knowledge Extraction*, vol. 7, no. 3, p. 97, 2025. doi: 10.3390/make7030097.

[5] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024. doi: 10.1109/TSE.2023.3341416.

[6] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "ChatGPT vs. SBST: A Comparative Assessment of Unit Test Suite Generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 1340–1359, 2024.

[7] E. Arteca, S. Harner, M. Pradel, and F. Tip, "Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks," in *Proc. 44th IEEE/ACM Int. Conf. Software Engineering (ICSE)*, Pittsburgh, PA, USA, 2022, pp. 1494–1505.

[8] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "ChatUniTest: a ChatGPT-based automated unit test generation tool," arXiv preprint, arXiv:2305.04764, Jan. 2023. doi: 10.48550/arxiv.2305.04764.

[9] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models," in *Proc. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, 2023, pp. 919–931.

[10] J. Hu, Q. Zhang, and H. Yin, "Augmenting Greybox Fuzzing with Generative AI," 2023. Available: https://www.semanticscholar.org/paper/Augmenting-Greybox-Fuzzing-with-Generative-AI-Hu-Zhang/db329fd9eada8b2f6533272b6e210c212dcbeab4

[11] S. J. Putra, Y. Sugiarti, B. Y. Prayoga, D. W. Samudera, and D. Khairani, "Analysis of Strengths and Weaknesses of Software Testing Strategies: Systematic Literature Review," in *2023 11th Int. Conf. Cyber and IT Service Management (CITSM)*, Makassar, Indonesia, 2023, pp. 1–5. doi: 10.1109/CITSM60085.2023.10455226.