International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 3, May-June-2026, ISSN (Online): 2395-566X

The Evolution of Large Language Models in Software Testing and Quality Assurance: Toward Governed and Agent-Based Collaboration

Kiran Paul Kanikaram
Principal Engineer, Testing

Abstract — Software testing is among the most time-consuming and challenging stages of the development lifecycle to automate. Effective testing requires contextual awareness, including an understanding of code semantics and the identification of edge cases. The advancement of Large Language Models (LLMs) has enhanced the feasibility of automated testing by enabling unit, end-to-end, and exploratory testing as well as supporting a more agentic Quality Assurance (QA) process. The following paper will discuss the current use of large language models in software testing, with an emphasis on test case creation, bug detection, and self-healing automated systems based on natural language prompts. Although LLMs are providing novel ways to improve testing efficiency, there are certain obstacles that require special attention. They include incorrect output due to hallucinations, incomplete test coverage, and decreased reliability as the software grows. To better understand the obstacles, the example of a checkout module taken from the existing literature is discussed. The results show that while LLM-based testing methods can achieve useful test coverage, they do not necessarily outperform search-based methods. The analysis concludes that the value of LLMs in Quality Assurance is maximized through a human-in-the-loop approach, supported by a five-layer governance framework. As a result, the role of the QA professional is evolving toward that of a test-quality engineer and AI supervisor, requiring an expanded skillset.

Keywords— Large language models; software testing; quality assurance; test generation; agentic AI; human-in-the-loop; test automation; hallucination.

I. INTRODUCTION

Historically, testing activities have constituted a relatively small proportion of overall software engineering effort, leading to an underestimation of the significance of the testing phase. Nevertheless, insufficient testing remains a primary cause of production issues, security vulnerabilities, and reputational harm. Automation frameworks such as record-and-playback tools, keyword-driven frameworks, and search-based test generators have dominated for decades. Although these frameworks can execute tests, they lack the ability to comprehend the intended behavior of the code. Recent advancements in transformer-based Large Language Models (LLMs), trained on extensive code and natural language corpora, are rapidly transforming this landscape [1], [2]. LLMs, including those from the GPT family, Codex, and CodeT5, facilitate automated generation of unit tests from function signatures and docstrings, translation of manually written test cases in natural language into executable code, and diagnosis of issues by interpreting stack traces, all without requiring additional code to perform these tasks [1], [2].

The transition from laboratory research to industrial application has progressed rapidly. Recent case studies demonstrate that automated test planning, generation, and execution are now effectively utilized in the development of safety-critical systems, such as automotive software [3]. Commercial QA solutions currently offer agents that can generate, plan, and execute test cases without step-by-step human scripting. However, empirical evidence indicates that existing solutions for automating test processes exhibit considerable failure rates [4], [5], [7], [17]. Existing LLM capabilities for software testing and QA are integrated into a comprehensive lifecycle framework that encompasses unit-level generation through agentic, autonomous testing.

II. BACKGROUND AND RELATED WORK

Test generation using automated tools predates LLMs by decades. There is already an established set of search-based software testing tools, such as EvoSuite, which use evolutionary algorithms to optimize the structure-oriented coverage metrics, and they remain extremely good at doing precisely that. They consistently outperform LLMs at maximizing coverage in terms of lines of code and branches [5], [6]. However, their deficiency was never related to being ineffective at covering the code, but rather to the quality of tests generated: tests written by EvoSuite and similar tools were known to include vague variable names, insufficiently assertive

statements, and "test smells" that made maintenance difficult even for those passing [5].

LLMs entered this space from a different starting point. Rather than relying on hand-crafted heuristics or symbolic analysis, LLM-based test generation works by feeding the model source code and surrounding context and asking it to produce a test as an experienced programmer would, drawing on what it has learned about code structure and intent rather than a fixed set of rules. Prior tools such as ChatUniTest and ChatTester proved that general-purpose models can write compilable unit tests for Java methods with little support from specialized tooling. This success prompted a wave of benchmarks, TestEval and SWT-Bench among them, designed to measure how well LLMs could write tests that actually trigger bugs and achieve meaningful coverage, not just compile [8]. As the area matured, surveys began to map out the full testing pipeline and to surface recurring challenges such as hallucination and environmental dependence [1], [9].

Two defining trends characterize this emerging area. The first is a shift from single-shot generation to iterative, feedback-driven, execution-aware generation, where the agent incrementally improves candidate tests based on compiler output and coverage rather than generating the test only once [7], [10]. The second trend involves moving from task-specific testing tools to generic, multi-step agentic testing systems capable of planning, executing, and adapting the entire test cycle with minimal human intervention at each stage [3], [11], [12]. Both trends raise a fundamental question: when will QA organizations be able to reliably use LLM-generated testing output, and what governance structures will be required?

III. LLM CAPABILITIES THROUGHOUT THE QA LIFECYCLE

1. Unit Test and Test Case Generation

The most advanced application of LLMs in the QA space is unit-level test generation. Given the method signature, class context, and sometimes a natural language description, an LLM can generate a test with its assertions, setup, and teardown code. Extensive comparative experiments with several instruction-tuned models and many dozens of actual classes demonstrate that generated tests are on par with the traditional tools in terms of correctness and readability, but still are inferior to search-based tools such as EvoSuite in terms of coverage and have a considerable rate of test smells that require cleaning up before tests become production-ready [5].

2. Test Script Generation and Migration

Aside from unit tests, LLMs can be used to transform manual natural-language test cases into automation scripts and translate scripts between different frameworks, e.g., a legacy Selenium test suite into its equivalent in Playwright. This particular activity benefits from LLMs because it is a translation task between two structured representations of intent and has been reported to significantly reduce the scripting effort typically required to keep automation suites up to date with application changes [13].

3. Exploratory and Agentic Testing

Agentic testing represents the current capability frontier: a system in which an agentic testing platform plans the test approach, generates or selects test scenarios, executes them against the system under test, analyzes the results, and decides on the next step. It seems like this process does not resemble a linear program where everything is executed top-down, but an explorer solving a task. There is at least one automotive enterprise that has achieved full automation at this planning stage, using LLM-based agents throughout the entire process, including requirements gathering, software development, test execution, and test reporting, with minimal human intervention [3]. Also, commercial testing tools have converged on a similar solution, creating agents that heal themselves when locators break due to UI changes and modifying their testing approach accordingly [14].

4. Defect Detection, Triage, and Program Repair

LLMs also find applications in analyzing test execution results: they analyze failure logs, stack traces, and diffs, provide insights into possible root causes of failures, offer suggestions for fixing the issue, or generate a regression test after the issue has been fixed. Automated program repair studies demonstrate the effectiveness of LLM-driven solutions in identifying defects and implementing fixes, and an innovative benchmark has emerged that specifically targets test generation for unusual, rare code paths [15], [22].

5. Test Maintenance, Self-Healing, and Test Smell Repair

Maintenance of a test suite in terms of updating locators, assertions, and fixtures in case of an application evolution has historically taken a significant portion of QA time, regardless of the initial cost of generating the test code. Co-evolution strategies in which LLM test generation is coupled with test template repair have been shown to reduce compilation and runtime errors in generated test code by alternating between generation and repair stages of test creation [7]. Execution-aware methods, in which an LLM receives only a backward slice of the program rather than the entire file, have been shown

to reduce hallucinations and increase coverage during iterative regression test generation [10].

IV. AN ILLUSTRATIVE EXAMPLE: TESTING A CHECKOUT MODULE

Some examples to demonstrate this are as follows: A mid-size engineering team has requested software to generate unit test cases using the LLM approach for the CheckoutService class of an e-commerce application. The CheckoutService class has an applyDiscount method that uses loyaltyTier of the Customer object and subtotal of the Cart object.

```
public class Checkout {
public BigDecimal applyDiscount(Customer customer, Cart cart) {
if (customer.getLoyaltyTier() == LoyaltyTier.gold) {
return cart.getSubtotal().multiply(new BigDecimal("0.90"));
}
return cart.getSubtotal();
}
}
```

Figure 1. Simplified target class for illustrative LLM-based test generation.

Based solely on the method signature, the LLM-generated test infers that a gold-tier customer and a non-zero subtotal should be provided, constructing an assertion that compares the discounted amount to the expected output. This example illustrates the LLM test generator's ability to infer intent from names and structures even when explicit specifications are absent. This observation is consistent with findings that LLM-generated JUnit tests increase line and branch coverage compared to the absence of tests, with line coverage ranging from the high 60s to the high 80s and branch coverage from the high 60s to the low 90s on benchmark classes [6].

Suppose the LLM is asked to extend coverage to a related Refund class that references customer.getRefundEligible(), a field that does not exist on the Customer class, possibly due to a prior refactor. Many LLMs will still generate a test that calls this method, resulting in a compilation failure rather than a valid test result. Recent research refers to this phenomenon as Field Access Hallucination, which accounts for a significant proportion of LLM-generated test failures. This issue has motivated the development of static-analysis-guided repair pipelines, which provide the model with only verified field-initialization context before test generation [4].

This scenario also highlights a more fragile yet important failure mode: brittleness under code evolution. For instance, if the discount logic is modified so that gold-tier customers no longer require a minimum cart balance to qualify—a semantically significant change implemented through a minor code edit—empirical assessments indicate that LLM-based test generation performs well on unmodified programs but declines sharply in pass rate when the underlying logic changes. Notably, most tests that fail with the modified code still pass with the original version, suggesting that the model often relies on pattern matching of familiar code frameworks rather than reasoning about updated behavior [17]. For QA organizations, this finding implies that an LLM-generated [21] suite may fail to detect the behavioral changes it is intended to identify.

V. EMPIRICAL EVIDENCE: WHAT THE NUMBERS SHOW

The example above conforms to and is based on a mounting amount of quantitative empirical evidence. Figure 1 summarizes some of the representative numbers reported in scientific articles cited in this paper.

Table 1. Representative quantitative findings from the LLM-based software testing literature.

Metrics	Reported Range/Findings	Source
Line coverage, LLM-generated JUnit tests (HumanEval benchmark)	67.0% – 87.7%, vs. 96.1% for EvoSuite and 88.5% for manual tests	[6]
Branch coverage, LLM-generated JUnit tests	69.3% – 92.8%, vs. 94.3% for EvoSuite	[6]
Tests generated in large-scale comparative study	216,300 tests across 690 Java classes from 4 LLMs and 5 prompting techniques	[5]
Baseline coverage before code evolution	79.2% line / 76.1% branch coverage, fully passing suites	[17]
Coverage after semantically altered code (SAC)	Pass rate falls to 66.5%; branch coverage falls to 60.6%	[17]
False-failure rate under SAC	>99% of failing tests still pass on the original, unmodified program	[17]
Multi-step agentic task failure rate	Approximately 70% in simulation testing	[18]

without structured oversight		
Projected enterprise agentic-AI adoption in IT operations	From under 5% (2025) to roughly 70% (2029)	[18]

Two prominent trends are apparent in light of the research findings discussed above. Firstly, LLM-powered testing is not inferior to conventional software testing tools in terms of classic structural coverage, nor does it outperform them; it can be used as an additional tool to increase coverage rather than replace existing solutions [5], [6]. Secondly, perhaps most importantly for production safety, the effectiveness of LLM-powered testing depends heavily on the similarity between the target codebase and the examples used during training. This similarity decreases over time as the code evolves, something conventional coverage benchmarks fail to take into account [17].

VI. CONSTRAINTS AND DANGERS

Four risk categories recur across the literature and merit explicit attention from QA leadership.

1. Hallucination
An LLM could produce references that are syntactically correct but semantically incorrect by referring to topics, techniques, or API behavior that do not exist, leading to errors at compile time or execution rather than useful test results. The above problem is similar to the hallucination problem seen in many LLM applications, where LLMs tend to generate false information outside their training distribution [4], [19].

2. Coverage Plateaus and Test Smells
Iteration in LLM test generation typically reaches a dead end: Each new prompt results in similar tests for the same covered code paths instead of making progress toward code paths yet to be covered, an issue that differs from but bears resemblance to mode failure in other kinds of test generation [10]. Tests generated show a high prevalence of test smells such as weak assertions, unclear names, and redundancy [5].

3. Brittleness Under Code Evolution
As demonstrated in Section 4 and Table 1, LLM-generated tests degrade disproportionately when source code changes semantically, implying that current models rely more heavily on surface pattern recognition from training data than on causal reasoning about program behavior [17]. This has clear effects on regression-testing strategy, since regression suites exist specifically to catch the kinds of behavioral changes LLMs are shown to be least reliable at detecting. [16]

4. Trust, Accountability, and Oversight Gaps
As test system design shifts from simple to multi-step agent creation, the issue of accountability becomes more pressing, as the failure of a self-driven agent to detect a defect raises questions about responsibility and the need to detect the failure before the process goes into production. Studies of industrial practice point to the lack of human intervention as the main cause of failure in multi-step agent designs, with simulations showing a nearly 70% failure rate in such processes [18].

VII. TOWARD TRUSTWORTHY ADOPTION: A FIVE-LAYER GOVERNANCE FRAMEWORK

To exploit the productivity gains discussed in Sections 3 to 5 while managing the risks outlined in Section 6, it is necessary to consider LLM-powered testing not as an alternative to manual testers and deterministic automation, but as a governed capability. This five-level model provides a structured approach based on common recommendations from the literature.

- Layer 1: Autonomy hierarchy: Categorize testing tasks based on reversibility as well as blast radius (for example, generating a unit test versus autonomously changing production test data fixtures) and allocate the right amount of autonomy, with complete autonomy being reserved only for low-risk and reversible tasks [18], [20].
- Layer 2: Static verification gate: Perform the tests generated by performing static analysis to identify access hallucinations to the fields and methods before execution and spending compute cycles or human effort, similar to static-guided repair-based approaches that have proven successful in reducing such hallucinations [4].
- Layer 3: Execution and coverage feedback cycle: Recognize the importance of single-generation approach to be just a draft rather than a finished product, and iteratively repair it through compiler outputs and coverage delta information, like co-evolution and execution-aware methods which have succeeded over one-time generation [7], [10].
- Level 4: Human checkpoints at critical thresholds: Require explicit human evaluation prior to integrating LLM-generated test cases into regression suites that are responsible for gating the release process, and prior to allowing any autonomous agent activity involving production write-access or irreversible actions [18], [20].
- Level 5: Ongoing audit and drift management: Evaluate LLM-generated suites against an evolved version of the code under test at regular intervals. As was discussed earlier in Section 6.3, point-in-time coverage

measurements may hide vulnerabilities that become apparent as the code evolves [17].

This hierarchy is intended to be independent of any vendor or model, and to plug into existing quality assurance governance practices without replacing them. The underlying assumption, as argued by the literature surveyed in this article, is that human oversight must be exercised at critical junctures of irreversibility or consequence, rather than uniformly across all levels of the generation process [20].

VIII. DISCUSSION: SIGNIFICANCE FOR QA PRACTICE

However, based on the evidence presented in the paper under consideration, an equity view of the use of LLMs in testing is well justified. Indeed, such models help reduce the cost of creating the first test draft, shift from manual to automated testing, and prioritize failures. The early experience with LLM testing in enterprises shows that these advantages apply to all aspects of testing, not just isolated actions [3]. At the same time, the presented evidence shows that LLMs do not replace QA engineers' expertise regarding test goals, failure relevance, and the passivity of the test suite after system modifications.

For QA companies, the major impact is more a change in the way human expertise should be used, rather than a reduction in the importance of such expertise. The most useful tasks for QA specialists have become those associated with work similar to test-quality engineers and AI governors: setting the risk tolerance for autonomous test generation, setting up static and dynamic checkpoints as discussed in Chapter 7, and keeping sufficient expertise in order to recognize when an autonomous test is technically sound but logically wrong. Such a development reflects broader industry perceptions that QA specialists are moving from scriptwriting to defining quality goals and controlling AI-generated tests [14], [20].

It will also affect skill acquisition and hiring practices. Prompt design skills, knowledge of the failure modes listed in Section 6, and verification gate design skills for generative models are as important for senior QA positions as scripting skills used to be. Companies that use LLMs for testing solely as a cost-saving measure without developing governance skills are at high risk of being exposed to brittleness and lack of trust described in the literature overview.

Summary and Future Work

In this way, Large Language Models are taking software testing out of a world where software tests were defined using deterministic procedures and blind search, and placing it into one where a system can read some code, determine what the developer intended, and write a reasonable test without any specification at all. Through an analysis of existing literature and a case study example that I provided, this article has demonstrated that while the above competency is real, it is highly inconsistent, excelling at generating drafts and translating languages while falling behind on structural coverage when compared to more mature search-based solutions, and being highly unstable when the code to be tested changes in a way that requires semantic rather than structural understanding.

The five-level approach described in Section 7 represents one possible way to achieve the observed benefits without falling victim to the risks identified in the literature. Future studies must focus on conducting longitudinal research in order to assess the effectiveness of LLM-produced test suite reliability across multiple releases rather than one-time evaluations, benchmarks specifically designed to assess the ability of code to evolve semantically over time rather than simple coverage, and empirical study of the administration mechanisms – determining whether human-based checkpoints actually help reduce the defect escape rate of the tests generated by LLMs and executed by agents. As the role of the QA engineer evolves into an overseer, so should the methods of evaluation used in the field evolve from simple pass/fail to the meaningfulness of the tests in six months and after several refactorings.

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. 4, pp. 911–936, 2024.
2. Q. Wang, J. Wang, M. Li, Y. Wang, and Z. Liu, “A roadmap for software testing in open-collaborative and AI-powered era,” ACM Transactions on Software Engineering and Methodology, vol. 34, no. 5, pp. 1–17, 2025.
3. S. Wang, Y. Yu, R. Feldt, and D. Parthasarathy, “Automating a complete software test process using LLMs: An automotive case study,” in Proc. 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 2025, pp. 373–384.
4. X. Guo and T. Tsuchiya, “Diagnosing and repairing field access hallucinations in LLM-based test generation,” in Software and Data Engineering, Springer, 2026.
5. W. C. Ouédraogo, K. Kaboré, H. Tian, et al., “Large-scale, independent and comprehensive study of the power of LLMs for test case generation,” arXiv preprint arXiv:2407.00225, 2024.

6. Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Using large language models to generate JUnit tests: An empirical study," arXiv preprint arXiv:2305.00418, 2023.
7. S. Gu, Q. Zhang, K. Li, C. Fang, F. Tian, L. Zhu, J. Zhou, and Z. Chen, "TestART: Improving LLM-based unit testing via co-evolution of automated generation and repair iteration," arXiv preprint arXiv:2408.03095, 2024.
8. W. Wang, K. Liu, A. R. Chen, G. Li, Z. Jin, G. Huang, and L. Ma, "Python symbolic execution with LLM-powered code generation," arXiv preprint arXiv:2409.09271, 2024.
9. C. Augusto, A. Bertolino, G. De Angelis, F. Lonetti, and J. Morán, "Large language models for software testing: A research roadmap," arXiv preprint arXiv:2509.25043, 2025.
10. C. C. Le, C. D. Van, T. D. Vu, T. M. Pham Vu, H. N. Phan, H. N. Phan, and T. N. Nguyen, "TestWeaver: Execution-aware, feedback-driven regression testing generation with large language models," arXiv preprint arXiv:2508.01255, 2025.
11. QualiZeal, "The rise of agentic AI: Transforming software testing in 2025 and beyond," QualiZeal Insights, 2025.
12. TestMu AI, "AI testing agents: Redefining software quality and QA in 2026," TestMu AI Learning Hub, 2026.
13. S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, "LLM for test script generation and migration: Challenges, capabilities, and opportunities," in Proc. 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), 2023, pp. 206–217.
14. Tricentis, "QA trends for 2026: AI, agents, and the future of testing," Tricentis Blog, 2026.
15. J. Zhang, Y. Liu, P. Nie, J. J. Li, and M. Gligoric, "exLong: Generating exceptional behavior tests with large language models," in Proc. 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 2025, pp. 1462–1474.
16. K. P. Kanikaram, "Transforming Software Quality Assurance through Artificial Intelligence, Automated Testing, and Intelligent Software Analytics," American International Journal of Computer Science and Technology, vol. 7, pp. 103–114, 2025, doi: https://doi.org/10.63282/3117-5481/aijcst-v7i5p109.
17. S. Haroon et al., "Evaluating LLM-based test generation under software evolution," arXiv preprint arXiv:2603.23443, 2026.
18. Elementum, "Human-in-the-loop AI agents: Deploying agentic AI with control," Elementum Blog, 2026.
19. M. Omar, V. Sorin, J. D. Collins, D. Reich, R. Freeman, N. Gavin, A. Charney, L. Stump, N. L. Bragazzi, G. N. Nadkarni, and E. Klang, "Multi-model assurance analysis showing large language models are highly vulnerable to adversarial hallucination attacks during clinical decision support," Communications Medicine, vol. 5, no. 1, Art. 330, 2025.
20. Galileo AI, "How to build human-in-the-loop oversight for AI agents," Galileo Blog, 2026.
21. K.P. Kanikaram, "AI-Based Regression Testing in Agile Software Development," International Journal of Management, vol. 16, 2026, Accessed: Jul. 02, 2026. [Online]. Available: https://ijmra.us/project%20doc/2026/IJME_JUNE2026/IJMIE2June266.pdf
22. Q. Zhang, C. Fang, S. Gu, Y. Shang, Z. Chen, and L. Xiao, "Large language models for unit testing: A systematic literature review," arXiv preprint arXiv:2506.15227, 2025.

AUTHOR BIOGRAPHY

Kiran Paul Kanikaram is a Principal Engineer, Testing at Majesco, with over 15 years of experience in software quality engineering across insurance, retirement, healthcare, and fintech. His research interests include large language models in software testing, AI-augmented quality assurance, and human-in-the-loop oversight frameworks. He is a member of IEEE and ACM.