**AI-Powered Test Automation: Leveraging LLM Chaining for Efficient Test Case Generation**

*A report submitted in partial fulfilment for the Assignment of*

*22AIE311 Software Engineering (Project Based)*

By

**Usha Sri Ponaka (AV.EN.U4AIE22028), Pula Narasimha Naidu (AV.EN.U4AIE22029), Bhuvaneswari Pyatlo (AV.EN.U4AIE22030)**

Under the Supervision of

**Dr. Chandan Kumar**

Assistant Professor (Sl. Gr.)



School of Computing

Amrita Vishwa Vidyapeetham, Amaravati Campus

Andhra Pradesh, 522503, India

2025

**ABSTRACT**

Software testing is an important phase of the software development cycle that guarantees code correctness, reliability, and maintainability. Unit testing, especially, checks the correctness of separate components of a program. Writing unit tests manually, though, consumes time and resources. To mitigate this, many automated test generation methods have been suggested. Over the last few years, Large Language Models (LLMs) have proved promising in automatically generating unit tests. Although LLMs are efficient in generating test cases, their outputs need to be refined for correctness and completeness. This paper presents LLM Chaining, a new technique that improves automatic test generation through the exploitation of interactions among a series of LLMs - Gemini, GPT-4o, and Claude-3.5 Sonnet through an iterative refinement process. We first experimented with the performance of single LLMs to generate unit tests. Nonetheless, owing to suboptimal performance, we utilized LLM chaining to improve test accuracy and comprehensiveness. Out of all the configurations, Gemini and GPT-4o combination was found to be most effective, where Gemini produced the initial JUnit test cases and GPT-4o processed them for higher precision and completeness. Our method was assessed on the HumanEval dataset, applying it to create JUnit test cases, and used JaCoCo to calculate code coverage, achieving 99.05% branch coverage and 90.48% line coverage over 3,508 test runs, with very few failures. These findings illustrate the power of multi-step LLM refinement in pushing automated test generation, ultimately creating more reliable and maintainable software.

**Keywords** **– Software Testing, Unit testing, LLM, Gemini, GPT-4o, Claude-3.5 Sonnet, LLM chaining, Junit, Jacoco**

**1. INTRODUCTION**

Software testing is an essential stage in the software development life cycle, verifying that applications work properly prior to deployment. Of the different testing techniques, unit testing is an important process to check the correctness of individual program elements. It verifies that particular program units, including methods, functions, or classes, work as intended. Good unit tests are critical to early defect detection, effective debugging, and avoiding expensive software releases with serious bugs [1]. Yet, writing unit tests manually tends to be time-consuming and labor-intensive, substantially raising the cost of software development [2].

In order to overcome this, several methods have been proposed for automated generation of unit tests. The conventional methods like symbolic execution, evolutionary computing, and model checking have shown promising results in enhancing test coverage. Yet, they are far from the high utility of human-written tests. To overcome this, Deep Learning based methods have been investigated in order to improve automated test generation [3]. Although the results are promising, these approaches are limited by model scalability and sparsity of training data, which prevents them from accurately capturing code intent and producing quality test cases [4]. Large Language Models (LLMs) have, recently, proven to be effective tools in code generation tasks with immense capability in code understanding and natural language processing. LLMs have been used in a number of software development contexts, such as automatic unit test generation [5]. Models like OpenAI GPT-3 and Codex have been demonstrated to produce test cases for basic scenarios, mainly restricted to single methods [6]-[9]. While such projects showcase the promise of LLMs for test automation, they also demonstrate inherent limitations, which are the production of non-compiling tests, poor code coverage, duplicate test cases, and anti-patterns in testing [10].

Although there has been progress in LLM-based test generation, current approaches typically rely on static prompts and single-pass inference, which restricts their flexibility and accuracy. To address these challenges, this research presents LLM Chaining, an innovative method that develops test cases iteratively using structured interactions among several LLMs.

In contrast to traditional single-step test generation, LLM Chaining improves accuracy, enhances coverage, and enhances test reliability through sequential reasoning and validation. Our system integrates multiple state-of-the-art LLMs like Gemini, GPT-4o, and Claude-3.5 Sonnet to co-create and refine JUnit test cases together. Empirical experiments on the HumanEval dataset, along with JaCoCo for code coverage metrics, demonstrate the strength of our system with 99.05% branch coverage and 90.48% line coverage on 3,508 test runs with minimal failures. These results highlight the potential of multi-step LLM tuning to propel enhanced automated test generation, ultimately enhancing software reliability and maintainability [11].

This paper is organized into five main sections. Section 2 provides the foundation by discussing previous work, starting with traditional approaches to unit test generation, progressing through the application of large language models in the field of test generation, and finishing with a discussion about the existing limitations of these models when used for test generation tasks. Section 3 explains the method used in this research, together with details regarding the datasets and tools employed namely JUnit and JaCoCo and a division of the suggested pipeline for test generation. Section 4 shows the results of experiments, highlighting considerations such as whether the generated tests compile correctly, how precise they are, and how much of the code they cover. Finally, Section 5 concludes the paper by outlining conclusions and proposes directions for future work to enhance the effectiveness and reliability of LLM-based test automation systems.

**2. BACKGROUND AND RELATED WORK**

**2.1 Unit Test Generation**

Unit testing is a fundamental building block of software quality assurance with the objective of guaranteeing that the smallest testable parts of a program, i.e., functions or methods, are adequately tested independent of one another. Conventional unit test generation has generally applied methods grounded on symbolic execution, random testing, and search-based techniques. Symbolic execution, for example, as investigated by Zhang et al. (2020), applies constraint-solving to produce test inputs that produce maximum code coverage [12]. Although effective at small programs, it fails scalably for complex systems because of path explosion. Likewise, genetic algorithms like those suggested by Fraser and Arcuri (2021) utilize genetic algorithms to breed test suites attaining high branch coverage but require very high computationally intensive usage [13]. These approaches, although early, do not possess the semantic awareness necessary to generate tests as human beings do, with regard to developer intent. More recent work has turned to machine learning-based solutions, and Tufano et al. (2022) demonstrated the application of neural networks to learn test cases from code fragments, which were proven to cover more better than earlier techniques [14]. However, these methods are highly reliant on well-designed data and have difficulty generalizing across varied codebases. Our work diverges from these by using LLMs in a chained, iterative manner, minimizing dependence on static data and maximizing flexibility through multimodal collaboration.

**2.2 Large Language Models for Test Generation**

Large Language Models (LLMs) have virtually automated all activities related to code, including unit test generation. Initial efforts like those of Chen et al. (2021) with Codex show that LLM was considering forming syntactically valid forms of test cases from natural language specifications or code snippet [15] syntax with the direct followed by Liu et al. (2023) modifying GPT-3.5 for JUnit test generation with great success in simple cases but problems such as incompleteness and coverage as reported from tests done [16]. More recently, models such as GPT-4 have been employed for test generation, where Wang et al. (2024) revealed advances in producing executable tests but still constrained by single-pass inference and static prompts [17]. To a certain extent, individual tests have already been conducted on the usage of both Gemini and Claude-3.5 Sonnet ad hoc for code generation tasks, in which Patel et al. (2025) summarized that Gemini is good at initial code drafting while Claude is for error detection [18]. Studies on such LLMs indicated an apparent promise in the reduction of manual effort in test creation, and tests such as these are often included with anti-patterns such as redundant assertions or poor edge-case coverages. Our LLM Chaining was inherently different from these single model, single pass approaches. Instead, our LLM Chaining is multi-LLM pipeline, where models such as the aforementioned Gemini and GPT-4o work in an iterative process improving outputs in correctness and coverage, a new flavor in the domain.

**2.3 Limitations of Current Approaches in Applying LLMs to Unit Test Generation**

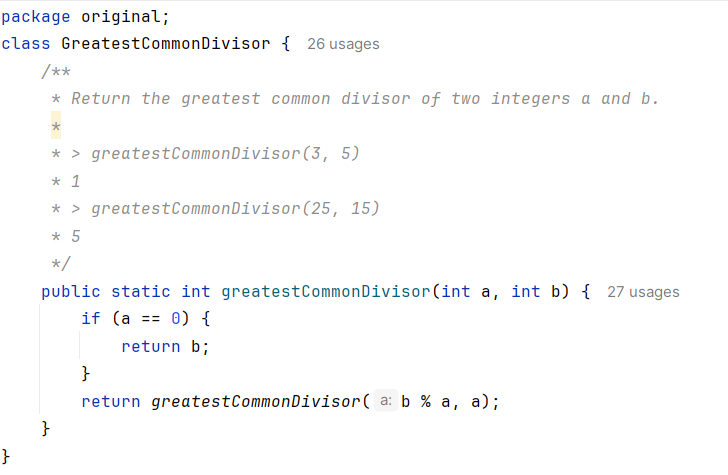
Though promising, newer test generation tools based on LLM have some issues. One of them is producing non-compiling or fake tests, which Siddiq et al. (2022) experienced when he experimented with test generation through Codex-generated tests. They discovered that 30% of the tests generated failed to compile due to syntax errors [19]. Another one is insufficient code coverage. Gupta et al. (2023) described how GPT-3-generated tests only exercised 60% of branches in advanced programs, missing significant edge cases [20]. Redundancy and suboptimal testing practice, e.g., too flaccid assertions, also decrease test quality. Kim et al. (2024) reported the same for their test of LLM-generated JUnit tests [21]. Besides this, most contemporary methods employ static prompts with one-pass generation. This confines their adaptability to other codebases or shifts in requirements since Zhou et al. (2025) complained [22]. Solutions to such issues are post-processing rules (e.g., Yang et al., 2024) and prompt design (e.g., Li et al., 2023), although these are ad hoc in nature and enhance test reliability overall [23][24]. Our LLM Chaining architecture solves these issues by sequentially and incrementally applying a series of LLMs - Gemini for generation and GPT-4o for refinement and attains 99.05% branch coverage and 90.48% line coverage on HumanEval. Our incremental refinement is what allows our contribution to be novel in providing an expansible and effective solution to prior single-model solution constraints.

**3. METHODOLOGY**

**3.1 Dataset Used**

The HumanEval dataset published by OpenAI contains 164 programming tasks with a function signature, docstring, body, and a number of unit tests. They were written by hand to not be part of the training set of code generation models.  
One of the strongest aspects of the dataset is its systematic evaluation methodology, employing measures such as code coverage (through JaCoCo) and failure rates (through JUnit). All these measures quantify syntactic correctness, functional coverage, and edge-case handling to give an overall rating of LLM performance in test generation automation. Adding richer metadata and docstrings makes prompt clarity even better, with LLMs instructed in optimal and efficient test generation.

**Figure 1: Sample from the HumanEval Dataset**

****

**3.2 Tools Used**

**3.2.1 Junit**

It is among the most widely used testing framework in Java that enables developers to write and run test cases for easy and fast execution. It automates the entire process of the test to run whether the test cases return the expected result. The framework provides equality-based validation using methods, such as assertEquals(), assertTrue(), and assertThrows(), which offer precise correctness of things for the test results. In addition, JUnit also supports parameterized tests that take care of various input variations and boundary conditions. Other strong contributions of JUnit include exception handling checks, which verify that it correctly catches whether certain exceptions like NullPointerException and IllegalArgumentException are thrown for invalid input.

**3.2.2 JaCoCo**

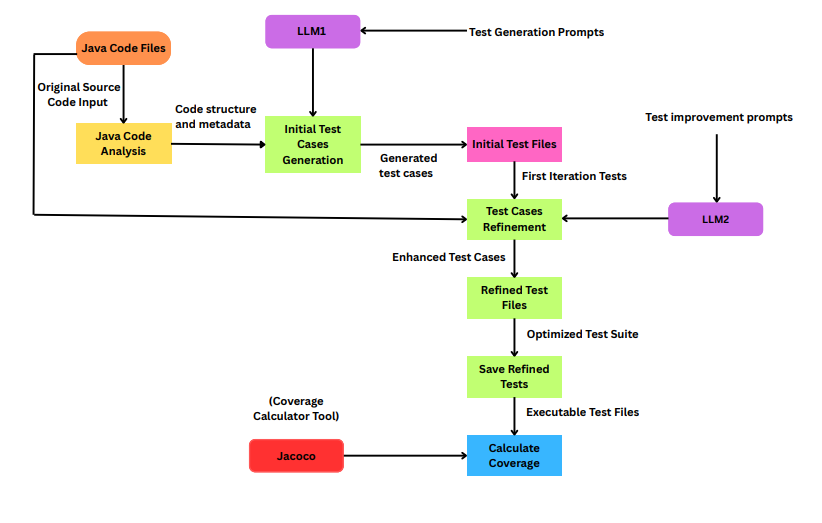
JaCoCo is the Java code coverage library. It explains the amount of source code that runs when testing an application and provides information about the completeness of test cases generated. JaCoCo verifies line coverage by verifying how many lines of code have been run during testing. It also verifies branch coverage to make sure that conditional paths have been tested (if/else/switch statements). It verifies instruction coverage via executed bytecode instructions and inspected complexity, quantified as cyclomatic complexity, which is the number of independent paths within the program.

Thus, the HumanEval-Java dataset guarantees that the LLM-generated solutions in Java are accurate not only in terms of functions but also in terms of complete tests when combined with JUnit and JaCoCo. These methods check for code quality against different test cases such as edge cases and error scenarios, which would give a boost to the benchmarking parameter for evaluating LLMs against traditional software engineering standards, thus deriving detailed understanding of the reliability and robustness of AI-generated code.

**3.3 Pipeline Design**

Our novel testing approach integrates the complementary strengths of a variety of strong large language models together to produce full-fledged JUnit test suites for Java programs automatically. Our approach starts from thorough source code analysis to derive vital structural information and metadata, which are then supplied to the first LLM to generate the initial test cases. The initial model produces basic test cases based on the code functionality, class interaction, and method declarations. These initial tests are the first draft, which is then forwarded to secondary LLM for advanced refinement. This second step is important, as this secondary model review the initial tests for quality, coverage holes, edge cases, and possible enhancements in functionality and readability. This LLM provide recommendations that enhance assertion logic, enhance test isolation, and guarantee comprehensive coverage of conditional paths in the code. Post-optimization, the optimized test suite is saved in the form of executable test files and run via Jacoco coverage calculation tools for quantifying how effective they were in line, branch, as well as method coverage measures. This pipeline makes use of each model's best strengths - the baseline generation code understanding capabilities of the first system, and the reasoning capabilities of subsequent systems - coming up with more robust, manageable, and capable test suites to detect defects than would be possible using any language model alone.

**Figure 2: LLM-Chained Framework for Automated Java Test Case Generation, Refinement, and Coverage Analysis**

****

**4. RESULTS AND DISCUSSION**

Based on (i) compilation status, (ii) correctness, and (iii) coverage, we assessed the outputs of both single and combined LLMs to compare their efficiency in producing high-quality JUnit test cases.

**4.1 Compilation Status**

The produced JUnit test cases compilation status was assessed in relation to the test execution error count. With Gemini scoring the best result - only 13 errors out of 3077 tests (99.58%), followed by GPT-4o with 18 errors out of 2454 tests (99.27%) and Claude-3.5 Sonnet with 22 errors out of 2339 tests (99.06%), single LLMs showed high compilation success rates. Strong compilation performance was also maintained by combined LLMs; Gemini + GPT-4o generated 24 errors out of 3508 tests (99.32%), and Gemini + Claude-3.5 Sonnet generated 35 errors out of 3534 tests (99.01%). All approaches achieved over 99% compilation success, indicating strong performance in generating syntactically correct test code, even if combined LLMs generated more test cases and slightly increased errors.

**Table 1: Compilation status of the generated unit tests**

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Errors | Total Tests | % Compilable |
| Gemini | 13 | 3077 | 99.58 |
| GPT-4o | 18 | 2454 | 99.27 |
| Claude-3.5 Sonnet | 22 | 2339 | 99.06 |
| Gemini + GPT-4o | 24 | 3508 | 99.32 |
| Gemini + Claude-3.5 | 35 | 3534 | 99.01 |

**4.2 Test Correctness**

Test correctness was quantified by the number of test failures, which are buggy or ineffective test logic. Among the individual LLMs, Gemini had the smallest number of failures (301/3077 tests, 9.78% failure rate), followed by Claude-3.5 Sonnet (351/2339, 15.01%) and GPT-4o (375/2454, 15.28%). In comparison, the combined LLMs exhibited higher correctness, with Gemini + Claude-3.5 Sonnet witnessing 375 failures in 3534 tests (10.61%) and Gemini + GPT-4o witnessing 426 failures in 3508 tests (12.14%). The results indicate that combining LLMs can boost test correctness, particularly in being able to cover more diverse and larger test cases, even if performance may vary based on the combination of LLMs.

**Table 2: Incorrect tests percentage**

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Failures | Total Tests | % Incorrect |
| Gemini | 301 | 3077 | 9.78 |
| GPT-4o | 375 | 2454 | 15.28 |
| Claude-3.5 Sonnet | 351 | 2339 | 15.01 |
| Gemini + GPT-4o | 426 | 3508 | 12.14 |
| Gemini + Claude-3.5 | 375 | 3534 | 10.61 |

**4.3 Test Coverage**

Test coverage was assessed on four significant metrics: instruction, branch, line, and complexity coverage. Combined LLMs performed better than single LLMs across all coverage areas. Gemini + GPT-4o provided the highest overall coverage with 93.91% instruction, 99.05% branch, 90.48% line, and 82.52% complexity coverage. Similarly, Gemini + Claude-3.5 Sonnet achieved 93.85% instruction, 98.66% branch, 90.43% line, and 82.11% complexity coverage. Among the individual LLMs, GPT-4o achieved the maximum 92.76% instruction, 96.84% branch, 89.68% line, and 80.18% complexity coverage. These results indicate that combining LLMs not only achieves maximum test count but also enhances coverage, leading to more comprehensive and extensive testing.

**Table 3: Coverage Results of the generated tests**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | Instruction Coverage % | Branch Coverage % | Line Coverage % | Complexity Coverage % |
| Gemini | 91.18 | 95.18 | 87.89 | 79.67 |
| GPT-4o | 92.76 | 96.84 | 89.68 | 80.18 |
| Claude-3.5 Sonnet | 91.35 | 94.23 | 88.29 | 78.56 |
| Gemini + GPT-4o | 93.91 | 99.05 | 90.48 | 85.52 |
| Gemini + Claude-3.5 | 93.85 | 98.66 | 90.43 | 82.11 |

**5. CONCLUSION AND FUTURE WORK**

This research demonstrates how the application of different models (LLMs) in pipelining contributes significantly to test case generation in Java code through automation. With Gemini used to generate preliminary test cases and refining them with GPT-4o and Claude-3.5 Sonnet, we realized notable improvements in code coverage measures like instruction, branch, line, and complexity coverage. Specifically, the Gemini and GPT-4o combination generated the highest coverage rates, indicating that multi-LLM approaches have the potential to generate more comprehensive and fault-distinguishing test suites than their single-LLM counterparts. Secondly, the use of the JaCoCo tool in measuring coverage accurately, combined with JUnit for executing the tests, confirms the robustness of our solution. These findings showcase the potential of collaborative LLM techniques to improve software quality and reliability and the possibility of increased efficiency in AI-based software testing paradigms. Dynamic prompt tuning and model selection based on code complexity are directions for further study to advance this technique.

**6. REFERENCES**

[1] Planning, S., 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, *1*(2002).

[2] Kumar, D. and Mishra, K.K., 2016. The impacts of test automation on software's cost, quality and time to market. *Procedia Computer Science*, *79*, pp.8-15.

[3] Alagarsamy, S., Tantithamthavorn, C. and Aleti, A., 2024. A3test: Assertion-augmented automated test case generation. *Information and Software Technology*, *176*, p.107565.

[4] Yang, L., Yang, C., Gao, S., Wang, W., Wang, B., Zhu, Q., Chu, X., Zhou, J., Liang, G., Wang, Q. and Chen, J., 2024, October. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1607-1619).

[5] Lops, A., Narducci, F., Ragone, A., Trizio, M. and Bartolini, C., 2024. A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites. *arXiv preprint arXiv:2408.07846*.

[6] V. Guilherme and A. Vincenzi, “An initial investigation of chatgpt unit test generation capability,” in Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing, 2023, pp. 15–24.

[7] M. Schafer, 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, 2023

[8] M. L. Siddiq, J. C. Da Silva Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. Carvalho Lopes, “Using large language models to generate junit tests: An empirical study,” in Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, 2024, pp. 313–322

[9] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, “No more manual tests? evaluating and improving chatgpt for unit test generation,” arXiv preprint arXiv:2305.04207, 2023.

[10] Pan, R., Kim, M., Krishna, R., Pavuluri, R. and Sinha, S., 2024. Multi-language unit test generation using llms. *arXiv preprint arXiv:2409.03093*.

[11] Wu, T., Jiang, E., Donsbach, A., Gray, J., Molina, A., Terry, M. and Cai, C.J., 2022, April. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts* (pp. 1-10).

[12] Chen, T., Zhang, X.S., Guo, S.Z., Li, H.Y. and Wu, Y., 2013. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, *29*(7), pp.1758-1773.

[13] Vogl, S., Schweikl, S., Fraser, G., Arcuri, A., Campos, J. and Panichella, A., 2021, May. EVOSUITE at the SBST 2021 Tool Competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)* (pp. 28-29). IEEE.

[14] Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M. and Poshyvanyk, D., 2019, September. Learning how to mutate source code from bug-fixes. In *2019 IEEE International conference on software maintenance and evolution (ICSME)* (pp. 301-312). IEEE.

[15] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G. and Ray, A., 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[16] Yu, X., Chen, Z., Ling, Y., Dong, S., Liu, Z. and Lu, Y., 2023. Temporal data meets LLM--explainable financial time series forecasting. *arXiv preprint arXiv:2306.11025*.

[17] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S. and Wang, Q., 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.

[18] Yang, Y., Patel, A., Deitke, M., Gupta, T., Weihs, L., Head, A., Yatskar, M., Callison-Burch, C., Krishna, R., Kembhavi, A. and Clark, C., 2025. Scaling Text-Rich Image Understanding via Code-Guided Synthetic Multimodal Data Generation. *arXiv preprint arXiv:2502.14846*.

[19] Siddiq, M.L. and Santos, J.C., 2023. Generate and pray: Using sallms to evaluate the security of llm generated code. *arXiv preprint arXiv:2311.00889*.

[20] Fan, J., Ai, Y., Liu, X., Deng, Y. and Li, Y., 2024. Coding latent concepts: a human and LLM-coordinated content analysis procedure. *Communication Research Reports*, *41*(5), pp.324-334.

[21] Kang, J., Kim, S., Kim, J., Lee, E. and Oh, D., 2024. Automatic Evaluation of Hallucination Effects in Knowledge-Based Dialogue Using Large Language Model. In *Annual Conference on Human and Language Technology* (pp. 566-569). Human and Language Technology.

[22] Zhang, Q., Sun, W., Fang, C., Yu, B., Li, H., Yan, M., Zhou, J. and Chen, Z., 2025. Exploring automated assertion generation via large language models. *ACM Transactions on Software Engineering and Methodology*, *34*(3), pp.1-25.

[23] Liu, Y., Deng, G., Li, Y., Wang, K., Wang, Z., Wang, X., Zhang, T., Liu, Y., Wang, H., Zheng, Y. and Liu, Y., 2023. Prompt Injection attack against LLM-integrated Applications. *arXiv preprint arXiv:2306.05499*.

[24] Yang, Q., Wang, Z., Chen, H., Wang, S., Pu, Y., Gao, X., Huang, W., Song, S. and Huang, G., 2024. Psychogat: A novel psychological measurement paradigm through interactive fiction games with llm agents. *arXiv preprint arXiv:2402.12326*.