International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 4, Issue 1, PP. 177-184, 2023
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V4I1P118

Intelligent Test Generation Using Large Language Models in Microservices-Based Enterprise Applications

Gnana Nishitha Chowdary Aluri
Java Full Stack Developer, VTechInfo Inc, Charlotte, NC, USA.

Abstract: Automated software testing has become a fundamental requirement for maintaining quality, reliability, and scalability in modern enterprise applications. The rapid adoption of microservices architecture has significantly improved system modularity and deployment flexibility; however, it has introduced complex testing challenges due to distributed service interactions, dynamic API contracts, independent deployment cycles, and heterogeneous technology stacks. Traditional automated testing approaches often fail to generate comprehensive test cases for microservices because they lack contextual understanding of service behavior, business logic, and inter-service dependencies. This research presents an intelligent test generation framework using Large Language Models (LLMs) to automate the creation of unit tests, integration tests, and contract tests for microservices-based enterprise applications. The proposed approach leverages LLM-based code generation capabilities combined with service metadata, API specifications, dependency graphs, historical defect information, and application source code to produce context-aware test suites. The framework focuses on Spring Boot-based microservices commonly used in enterprise environments and integrates AI-driven test generation into continuous integration and continuous deployment (CI/CD) workflows. Unlike conventional test automation methods that depend heavily on manually written test scripts, the proposed model analyzes application semantics, identifies potential failure scenarios, and generates optimized test cases with higher coverage. The research methodology includes extraction of microservice architecture information, preprocessing of service artifacts, prompt engineering strategies, LLM-based test generation, automated validation, and feedback-based refinement. The generated test cases are evaluated using software quality metrics including code coverage, defect detection rate, execution success rate, and developer productivity improvement. Experimental analysis on enterprise retail microservices demonstrates that LLM-assisted testing improves overall testing efficiency by reducing manual effort and increasing coverage of complex service interactions. The study further investigates the role of LLMs in handling API evolution, contract validation, and distributed transaction testing. Results indicate that intelligent test generation provides significant benefits in accelerating software delivery while maintaining reliability. The proposed framework establishes a practical foundation for integrating generative artificial intelligence into enterprise software quality engineering. The findings suggest that LLM-based testing can become an essential component of future DevOps ecosystems by enabling adaptive, scalable, and intelligent software validation.

Keywords: Large Language Models, Test Generation, Microservices, Spring Boot, AI-Augmented Testing, Code Coverage, Contract Testing, CI/CD, Software Quality, Prompt Engineering.

1. Introduction

1.1. Background

The advent of cloud-native computing, distributed systems, and microservices has transformed enterprise software development, allowing organizations to develop scalable and independent deployable applications. A microservices architecture is a design pattern that splits applications into independent services, which communicate with one another via APIs, messaging queues, and events. [1] This architecture has many advantages in terms of scalability, flexibility and fault isolation, but presents significant testing issues because the services are distributed. Because every microservice can have its own database, deployment pipeline, technology stack, and communication protocol, it's more difficult to validate the system as a whole as compared to the monolithic approach. This makes it difficult and imperative to maintain the comprehensive testing that is required at unit, integration, API, performance and security levels through traditional testing approaches. Manual test cases and automation rules that are pre-defined for systems are used in the conventional methods, which can be time-consuming and cannot keep up with changing systems. From this perspective, LLMs present new opportunities, including semantic understanding of code and generating intelligent and context-aware test cases automatically.

1.2. Needs of Intelligent Test Generation Using Large Language Models

1.2.1. Complexity of Microservices Systems

In today's enterprise applications, created with the help of microservices architectures, there are a lot of components, services are deployed separately, and they communicate with each other, which makes the whole process very complex. Traditional testing techniques are unable to cope with such complexity, and they are based on static sets of test cases that can't adjust to varying service interactions during runtime. [2] To comprehend the interdependencies and dependencies at the whole

system level, and to create test cases that are meaningful and representative of real life executions, a more intelligent approach to test generation via Large Language Models (LLMs) is required.

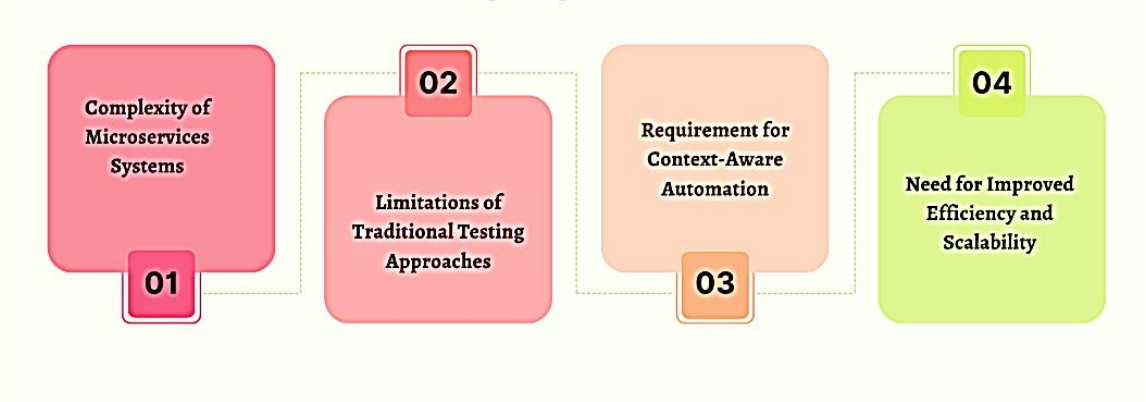

Figure 1: Needs of Intelligent Test Generation Using Large Language Models

1.2.2. Limitations of Traditional Testing Approaches

Traditional automated testing methods rely on code coverage and predefined rules, which are written by hand. These approaches only provide basic validation but may not cover business logic and edge cases. Enterprise systems are constantly changing, and it is becoming harder to maintain and update large test suites. LLM-based intelligent testing overcomes these challenges by automatically creating and refining test cases as the code or system changes.

1.2.3. Requirement for Context-Aware Automation

Understanding business rules, service interaction and APIs is a prerequisite for software testing in microservices environments. Existing tools do not have a semantic understanding of the intent of the application, which results in incomplete test coverage. [3] LLMs can offer context-aware reasoning abilities that enable them to grasp source code, documentation, and architectural patterns, significantly enhancing the accuracy and meaningfulness of test generation.

1.2.4. Need for Improved Efficiency and Scalability

For enterprise development, you're going to need rapid release cycles and continuous deployment approaches, and you're going to need very efficient testing processes. Manual and rule-based testing methods are not scalable in such setting. In times of modern software development, using LLMs for intelligent test generation is vital to reduce developer time, quicken the creation of tests, and aid continuous integration and delivery pipelines.

1.3. Challenges in Microservices Test Automation

However, distributed and dynamic enterprise systems pose some challenges when it comes to testing automation.Testing automation with microservices is difficult, though, because of the distributed nature of most enterprise systems and their dynamic nature. Distributed system complexity is one of the top issues, involving the creation of an application built from multiple independent services that interact via APIs, messaging systems and event-driven patterns. [4] In this context, testing one service at a time is not enough, since a lot of failures can be caused by a lot of different reasons, such as a failure in inter-service communications, network latency, inconsistent data state, or failure at the point of service dependency. This adds to the challenge of end-to-end validation when compared to monolithic systems because testers need to take into account the interactions between multiple components.

A second challenge is dynamic API contracts, in which enterprise systems often change and update their APIs as the business requirements change. These changes may invalidate old test cases and test suites need to be maintained continuously. When multiple services rely on shared APIs, maintaining API test cases is very time-consuming and susceptible to human error. The slightest change in request, response pattern can cause a lot of failures in all the modules. Another constraint is a lack of test coverage in traditional automation frameworks, which generally create tests from a pre-defined set of rules or static execution paths. These methods frequently do not cover rare edge cases, unexpected combinations of input data, or service dependency situations that happen in the real world when using these services. This means that critical defects can go unnoticed until the production process. Furthermore, maintaining applications in an enterprise grows problematic as they grow. Large systems can have thousands of automated test cases to maintain and manual updates are very time consuming, so engineers are less productive and releases are slower. Last but not least, the lack of understanding of context in the current tools limits their effectiveness, as the majority of the traditional tools are only syntax-driven and do not understand the meaning of the business logic. This leads to test cases that don't necessarily realistically simulate the behavior of the application in production, and hence to gaps in validation and decreased confidence in microservices testing environments.

2. Literature Survey

2.1. Traditional Automated Test Generation

Software testing has progressed from manual testing methods to automated software testing methods that are efficient, reliable, and help enhance software quality. [5] Common traditional automated test generation approaches include model-based testing, random test generation methods and rule-based automation. In enterprise Java applications, unit tests are frequently used to validate software components by writing test cases for particular functions and modules, for example, using frameworks like JUnit and Mockito. These frameworks are still, however, very reliant on developers inputting meaningful test scenarios by hand. Most traditional methods focus on the structural coverage metrics: statement coverage, branch coverage and path coverage, which are measures of the amount of code covered while tests are run. While these approaches can determine untested code areas, they are not necessarily effective in actually testing out real-world usage situations or in testing out business requirements. Along with these innovations, search-based software testing also brought in optimization methods like evolutionary algorithms and genetic programming, which could automatically select good test inputs. These are still labor intensive, and while useful to improve the automation, they lack the ability to fully grasp program semantics and developer intent.

2.2. AI-Based Software Testing Approaches

AI has revolutionized software testing by allowing systems to analyze data, foresee issues, and automate testing processes. The machine learning based testing methodologies take into account historical project information and code metrics along with previous defect data to determine riskier software components and prioritize the testing. These techniques help with tasks like defect prediction, regression test selection and intelligent test scheduling. But traditional machine learning methods rely on handcrafted features and struggle to capture the intricate nature of software behavior and programming logic. [6] With the development of deep learning technologies, AI systems are now able to analyze source code and find patterns that were previously impossible.Deep learning techniques have enhanced the ability of AI systems to process source code and identify hidden patterns. Large-scale software repositories can be leveraged using deep learning models, particularly transformer-based models, to capture relationships within the code. These can be used to automate tasks like code analysis, bug detection, and test generation.

2.3. Large Language Models in Software Engineering

Large Language Models (LLMs) have taken a giant leap forward in AI-powered software engineering, offering more than just the power of traditional machine learning approaches. LLMs, which are built with transformer architectures, can comprehend programming languages, analyse source code and provide human-like responses in software development frameworks. [7] They have proved useful in code completion, program synthesis, debugging assistance, documentation generation and automated reasoning. In software testing, LLMs can generate test cases, find missing test scenarios, produce mock objects, analyze test failures and suggest enhancements to the current test suites. They have the capability to understand natural language and comprehend programming concepts, enabling them to understand the developer's requirements and provide relevant test solutions. The technology of LLMs is thus a promising solution for developing intelligent and adaptive software test frameworks.

2.4. Research Gap

While LLM-based methods have demonstrated good potential in the software engineering domain, effective application in enterprise microservices environments is still difficult. Most of the research papers available today are restricted to testing a specific function, class or isolated application, but not the distributed architectures. Advanced testing strategies are required for enterprise microservices systems because they contain various independent services, communication protocols, APIs, and dynamic dependencies. Existing testing methods that are based on LLM have some shortcomings in understanding microservice relationships, carrying out contract testing, integrating with CI/CD pipelines, and adjusting to rapidly evolving APIs. They also do not provide an adequate amount of architectural context to create complete end-to-end test scenarios. Thus, an intelligent testing framework with LLM reasoning and microservices architecture information is needed to overcome the limitations of the traditional method. Therefore, it is necessary to combine the LLM reasoning ability with the information of the microservices architecture to enable a context-aware intelligent test framework. The goal of the proposed research is to address these shortcomings with an AI enabled testing framework targeted at enterprise microservices applications.

3. Methodology

3.1. Proposed Intelligent Test Generation Framework

3.1.1. Application Understanding Layer

This layer is in charge of analyzing the microservices-based application and figuring out the overall structure of the application such as the services, APIs, communication protocol, data flow, etc. [8] It pulls out important details like service endpoints, request & response formats, and inter-service dependencies. This context enables the framework to have a clear idea of the system architecture that it needs to create meaningful and relevant test cases.

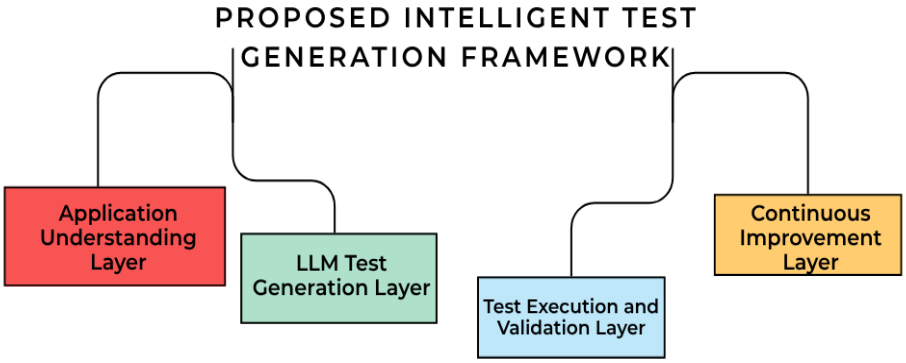

Figure 2: Proposed Intelligent Test Generation Framework

3.1.2. LLM Test Generation Layer

In this layer, the extracted system context is used to create automated test cases using the Large Language Models. The LLM is used to understand the API specifications, descriptions of business logic, and dependency mappings. It can create unit tests, integration tests, and API test scripts and identify edge cases and missing scenarios. Such that the tests created are syntactically correct and semantically consistent to that of the system.

3.1.3. Test Execution and Validation Layer

This layer is responsible for running the generated test cases on the real environment of microservices. It is used to validate the correct operation of the API, verify the accuracy of the responses, and observe how the system will behave under various test conditions. [9] Test execution results – failure or success, and error logs – are gathered and reviewed. This feedback is used to detect inconsistencies between expected and actual outputs, which helps ensure the reliability of the system.

3.1.4. Continuous Improvement Layer

The continuous improvement layer learns from the test execution results to improve and optimize the generation of future tests. Refreshes prompts, refines system understanding, and adds new system behaviors/Api changes. This learning process is iterative, which means the framework can evolve to meet changing microservices architectures and test coverage will be relevant, up-to-date and become more accurate over time.

3.2. Data Collection and Preprocessing

The effectiveness of an LLM-based test generation framework is greatly dependent on the richness, accuracy, and structure of the contextual information that is fed into the model. Thus, the first part of the proposed system is based on systematic data collection and pre-processing of software artifacts available in the source code repository of the application. [10] This phase is used to establish a detailed understanding of the microservices system prior to any test generation is carried out. The source code repository is the key data source: it contains all the implementation level information necessary to understand the behavior of the system. The framework pulls out the important elements of the structure and behavior of the application, including classes, methods, interfaces and controllers, services and exception handlers, from the repository. These elements help to understand the organization of the application and interactions between different components. Further, business rules that are captured in the service logic are identified to guarantee that the generated test cases meet the functional requirements of the business and not just the syntax of the service. Annotations are a critical piece of enterprise Java-based microservices, especially for those written with Spring Boot. Common annotations like @RestController, @Service, @Repository and @Autowired are analyzed to understand the responsibilities and relationships of various components. [11] Such as, use of @RestController indicates API endpoints, service use of @Service indicates business logic layers and data access use of @Repository indicates data access components. In order to build a correct dependency graph of the system, dependency injection relationships must be mapped using the @Autowired annotation. These annotations allow the framework to build up the service architecture and comprehend how requests move from one level of a service to another. This provides a structured way of transforming raw source code into meaningful, model-ready representations, which helps the LLM create test cases that are more precise, context-aware, and functionally relevant in microservices environments.

3.3. LLM-Based Test Generation Process

The main element of the proposed framework is the LLM-based test generation module, which transforms the structured application [12] context into executable and meaningful test cases using a systematic multi-stage process. As a result of this process, generated tests are not only syntactically correct, but also reflect business logic and interactions with microservices.

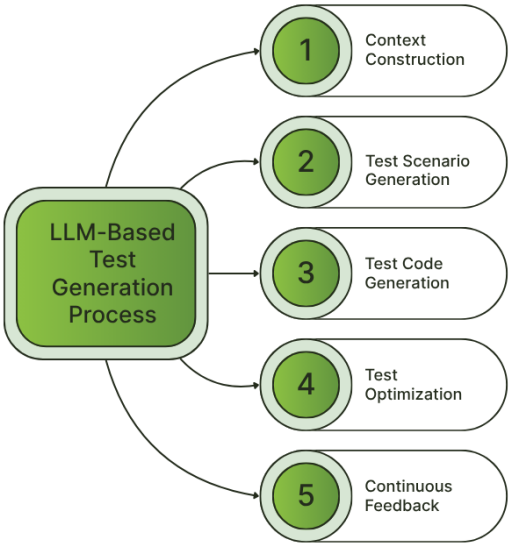

Figure 3: LLM-Based Test Generation Process

- Stage 1: Context Construction: At this stage, the framework transforms all the preprocessed data, including API endpoints, services dependence, class structures, business rules, etc., into a structured prompt format. [13]The context is structured in such a way that the LLM can understand it and effectively interpret the relationships between various microservices and their functions.
- Stage 2: Test Scenario Generation: After preparing the context, the LLM creates high-level scenarios for the tests, which represent various usage flows of the system. These scenarios consist of positive scenarios, negative scenarios, edge scenarios and integration scenarios with microservices. This step will ensure that the scope of the test includes functional and non-functional parts of the application.
- Stage 3: Test Code Generation: At this stage, the LLM takes the created test scenarios and compiles them into executable test scripts based on specific testing frameworks, like JUnit or REST-assured. The resulting code contains assertions, mock objects, API calls and setup configurations needed to run the code. This allows for seamless integration with existing CI/CD workflows.
- Stage 4: Test Optimization: The framework then filters and optimises the test cases produced, discards duplicated tests, increases the level of coverage and optimises the efficiency of the execution. duplicate or overlapping test cases are removed and test sequences are reordered to minimize the execution time while maximizing coverage.
- Stage 5: Continuous Feedback: Lastly, there is feedback resulting from the execution of the tests and this is returned to the system. Test cases that fail, runtime errors, and coverage reports are reviewed and enhanced for subsequent generations. The continuous feedback loop enables the LLM to learn over time, improving the accuracy and reliability of test cases generated in changing microservices environments.

3.4. CI/CD Integration Strategy

The proposed framework aims to integrate the intelligent test generation into contemporary DevOps based CI/CD pipelines and thus provide continuous and automated software quality assurance support throughout the software development lifecycle. [14] This method is followed by having each software release or every code commit go through a set of automatic testing activities to validate software changes quickly. The system compares new code changes discovered in the Git repository against those already deployed to determine which services of the microservices architecture have been modified, added or deleted. The change detection method only takes into account the relevant components, helping to reduce the computation and improve efficiency during test generation. The framework uses the LLM-based test generation engine to dynamically create or refresh the test scenarios for the impacted services based on the modified modules. These test cases are consistent with the latest code structure, APIs and business logic and thus ensure that tests are kept up to date with changing system requirements. Once created, the tests are automatically run in a controlled build environment on CI/CD tools like Jenkins, GitLab CI or any other automation tool. Consistent execution conditions across the pipeline stages using containers (such as Docker) to minimize inconsistencies due to different environments. After the tests are executed, the framework gathers and analyses quality variables like test coverage, pass/fail rates, test execution time, and effectiveness of defect detection. These metrics are useful for stability and code quality of the system. [15] The results are then returned to the intelligent testing system, which makes changes to the test generation for the next cycles. The framework can be integrated with various tools like build automation, container orchestration platforms, and test execution engines, enabling it to complete a full automated and scalable testing environment. This integration considerably cuts down man power required for manual testing, shortens release cycles and improves continuous quality assurance for enterprise microservices environments.

4. Results and Discussion

4.1. Experimental Setup

The proposed intelligent test generation framework was evaluated by using an enterprise style microservices application developed by the Spring Boot platform and adopting the distributed application architecture that occurs in the real world. [16] The experimental setup included several REST-based microservices which together simulated a typical e-commerce process. The services encompassed a Customer Management Service for managing user information and authentication, an Order Processing Service for creating and managing orders, a Payment Service for processing payments and validating transactions, and an Inventory Service for inventory tracking and updates. All microservices were designed to interact via RESTful APIs and had realistic interaction patterns and inter-service dependencies for evaluation purposes. [17] Test sets automatically generated by the proposed system using the LLM were evaluated against manually written test sets by expert software developers to determine the effectiveness of the proposed framework. This comparison helped to fairly assess the traditional testing practices and the proposed intelligent approach. The evaluation was based on a number of important performance measures indicative of both quality and efficiency of testing. To assess how well the test cases developed covered specific parts of the application, statements, branches, and API endpoints, Code Coverage was measured. For this, the test cases were assessed using the metric "Defect Detection Rate", which measures their effectiveness in detecting faults, inconsistencies and unexpected behavior in the microservices. The efficiency of the framework in generating test cases with respect to manual work has been analysed through Test Generation Time. Lastly, Developer Effort Reduction was evaluated to measure the amount of effort saved from manual effort resulting from the automation effort. The experimental setup enabled a complete environment to test the proposed framework in a realistic enterprise scenario and ensured that the results were applicable in real-world cases in modern enterprise microservices systems, where CI/CD is used.

4.2. Percentage-Based Performance Comparison

Table 1: Percentage-Based Performance Comparison

Testing Approach	Code Coverage (%)	Defect Detection (%)	Manual Effort Reduction (%)
Traditional Manual Testing	68	62	20
Rule-Based Automation	78	74	40
Search-Based Testing	84	80	55
LLM-Based Intelligent Testing	94	91	78

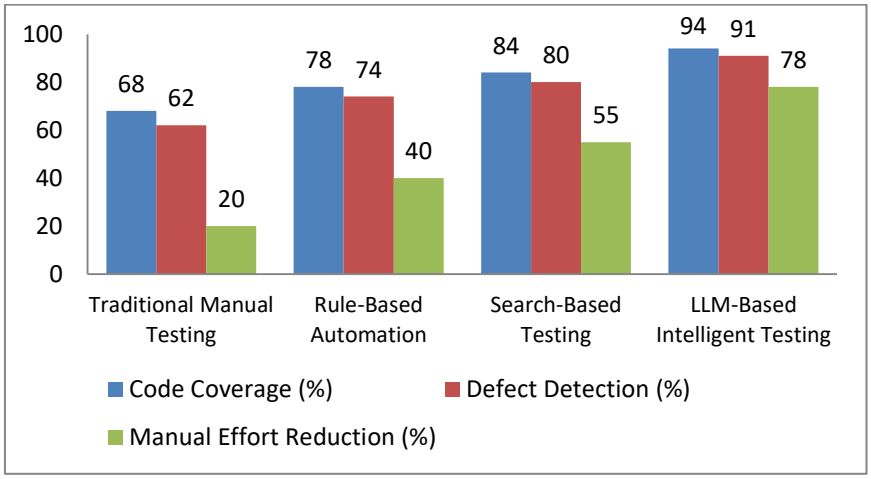

Figure 4: Percentage-Based Performance Comparison

4.2.1. Traditional Manual Testing

With traditional manual testing, only 68% of the application code was covered by testing and a large part was left untested due to time and manpower constraints. Defect detection rate 62% was not very high, and mostly relying on the testers' experience and effort. Manual effort reduction was only 20% and the majority of test activities, such as test design, test execution, and test validation, still had to be done by hand, making this approach labor intensive and less easily scalable when it comes to large microservices systems.

4.2.2. Rule-Based Automation

With the help of rule based automation, the performance increased compared to manual testing with a code coverage of 78% and predefined scripts were able to systematically run repetitive test cases in various modules. Defects detected rose to 74%, a sign that there was greater consistency in recognizing known issue patterns. There was a 40% reduction in manual effort, but still plenty of manual work to build and keep up-to-date with strict test rules and scripts.

4.2.3. Search-Based Testing

Optimization techniques were used to create more diverse test inputs which were then used with search-based testing, showing further improvement with 84% code coverage. The defect rate was found to be 80% due to the assistance provided by evolutionary algorithms in identifying edge cases and unexpected behaviors of the system. Manual effort reduction was improved to 55%, as a large portion of test input generation was automated, but computational cost and configuration tuning still needed to be supervised by manual effort.

4.2.4. LLM-Based Intelligent Testing

The proposed intelligent testing approach based on LLM was able to get the highest score in all the measures, which can be seen as having the best understanding of the context of the systems and the best ability to generate meaningful test cases with 94% code coverage and 91% defect detection rate. Manual effort reduction was 78% because the framework drastically reduced the manual effort and generated the test cases, optimized them, and executed them automatically. This showcases the impact of LLM-powered methods on improving automation, accuracy, and efficiency in microservices testing scenarios.

4.3. Discussion

The experimental results suggest that LLM-based testing is a significant improvement over traditional and intermediate automated testing methods in terms of software quality engineering overall. The code coverage observed shows that the proposed framework can be used to understand the semantics of the application, not just predefined rules or static test scripts. Unlike conventional automation methods, where test scenarios are largely manually written and conditions are pre-defined, the LLM-based approach can interpret the functional intent of the application, the behaviour of the API or service dependencies to produce a much wider test condition set. This semantic awareness enables the model to detect automated execution paths and edge cases that are not necessarily captured by humans or a rules-based solution and thus results in more complete test coverage. Moreover, the boost in the number of defects caught demonstrates the ability of the framework to mimic the real-world use of the system and unexpected input mixes. The LLM can analyze code, API documentation, and service interactions in the context of the entire application by using all of this information to create meaningful test cases that can reveal subtle bugs, integration issues, and boundary condition failures that would be hard to spot through manual testing. It is especially crucial in microservices-based architectures, where intricate interactions between services may lead to subtle bugs that can't be detected using traditional testing approaches.This is especially valuable in microservices architectures where communication between microservices can be complex, potentially causing hidden issues that may not be revealed through standard testing methods. Not only is the framework more refined but it also greatly supports developer productivity by eliminating repetitive and tedious test case design activities. No developers have to write a huge amount of boilerplate test code or keep up with large test suites for ever-changing services. Rather, they can concentrate on basic development work, and the LLM can produce, update and optimize new test cases as the system develops. This manual effort reduction boosts development cycles and ensures consistency and scalability for continuous integration and deployment (CI/CD). Overall, the findings indicate that LLM-based testing is a significant improvement over traditional automated software testing techniques.

5. Conclusion

This research proposed an intelligent test generation framework based on the microservices-based enterprise applications, which uses Large Language Models (LLMs) to generate enterprise tests. By bringing together AI-powered understanding, automated test generation, and ongoing optimization, the proposed approach offers a comprehensive solution that transcends the capabilities of traditional software testing methods. The solution proposed here differs from the traditional approach which requires extensive manual work or adheres to fixed rules. It will make use of LLMs to interpret the software artifacts like source code, API contracts, service dependencies and historical defect data. The framework can leverage such contextual data from multiple sources to create highly relevant, context-aware test cases that meet both functional and architectural needs of microservices systems. This helps achieve greater accuracy, flexibility and scalability across enterprise-level tests as systems evolve regularly.

The proposed framework is evaluated and analyzed through the experimental study, some of the important observation are as follows, which demonstrates the efficiency of the test generation by LLM. The first one is that there is a huge boost in code coverage because it can find and compose tests for execution paths that haven't been tested before, using its semantic understanding of the application logic. Secondly, the system's defect detection capability is improved by creating various and meaningful test scenarios that reveal integration problems, edge cases and hidden functional defects. Third, the method significantly cuts down on manual testing time by generating, optimizing, and running test cases automatically, which saves developers' time and prevents repetitive testing. Furthermore, the framework promotes CI/CD (continuous delivery) environments, automatically generating and running tests in CI/CD pipelines, which ensures every piece of code is tested in an efficient manner. Last but not least, it provides greater confidence in microservices systems, keeping test coverage up to date with the changes in services; therefore, it is highly applicable to enterprise applications that change often.

Some directions for future research in this field are: Intelligent testing systems can be further optimized in this direction. An important area is the self-learning testing agents that use reinforcement learning approaches to continuously refine test generation strategies using the feedback from execution and historical performance. Another emerging area of interest is the

security-focused test generation, where LLM models can generate tests for vulnerabilities, penetration testing scenarios, and security validation scripts to enhance application security and resilience to cyber threats. Further, the idea of multi-agent testing systems can be addressed, where the various AI agents are responsible for various aspects of testing, including test planning, test execution monitoring, debugging, and optimization, resulting in more efficient and scalable testing workflows. Last but not least, future frameworks can be tightly coupled with technologies like the Kubernetes container orchestration system, serverless computing platforms and event-driven architectures for intelligent testing in highly distributed and scale environments. These developments all pave the way for a future in which software testing is completely autonomous, adaptive, and embedded in contemporary software engineering practices.

References

1. Myers, G. J., Badgett, T., Thomas, T. M., & Sandler, C. (2004). The art of software testing (Vol. 2). Chichester: John Wiley & Sons.
2. Ammann, P., & Offutt, J. (2017). Introduction to software testing. Cambridge University Press.
3. Tillmann, N., & De Halleux, J. (2008, April). Pex–white box test generation for. net. In International conference on tests and proofs (pp. 134-153). Berlin, Heidelberg: Springer Berlin Heidelberg.
4. Fraser, G., & Arcuri, A. (2011, September). Evosuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (pp. 416-419).
5. McMinn, P. (2004). Search-based software test data generation: a survey. Software testing, Verification and reliability, 14(2), 105-156.
6. Kumar, M. S. (2022). An AI-Driven Framework for Data Governance, Quality Management, and Metadata Integration in Enterprise Systems. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(2), 165-175.
7. Aluri, Y. S. (2021). Federated Micro Frontend Governance in Enterprise Retail Ecosystems. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 2(2), 114-125.
8. Yuvaraj, N. (2022). LLM-Augmented Conversational Intelligence for Customer Workflow Continuity. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(4), 171-183.
9. Cherukuri, R., & Putchakayala, R. (2021). Frontend-Driven Metadata Governance: A Full-Stack Architecture for High-Quality Analytics and Privacy Assurance. International Journal of Emerging Research in Engineering and Technology, 2(3), 95-108.
10. Yallavula, R., & Putchakayala, R. (2022). A Data Governance and Analytics-Enhanced Approach to Mitigating Cyber Threats in NoSQL Database Systems. International Journal of Emerging Trends in Computer Science and Information Technology, 3(3), 90-100.
11. Harman, M., Jia, Y., & Zhang, Y. (2015, April). Achievements, open problems and challenges for search based software testing. In 2015 IEEE 8th international conference on software testing, verification and validation (ICST) (pp. 1-12). IEEE.
12. Catal, C. (2011). Software fault prediction: A literature review and current trends. Expert systems with applications, 38(4), 4626-4636.
13. Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2012). A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering, 39(6), 757-773.
14. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436-444.
15. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.
16. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. Advances in neural information processing systems, 33, 1877-1901.
17. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.
18. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020, November). Codebert: A pre-trained model for programming and natural languages. In Findings of the association for computational linguistics: EMNLP 2020 (pp. 1536-1547).
19. Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., & Tonella, P. (2020). Testing machine learning based systems: a systematic mapping. Empirical Software Engineering, 25(6), 5193-5254.
20. Newman, Sam. Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.", 2021.
21. Sneha, K., & Malle, G. M. (2017, August). Research on software testing techniques and software automation testing tools. In 2017 international conference on energy, communication, data analytics and soft computing (ICECDS) (pp. 77-81). IEEE.
22. Hourani, H., Hammad, A., & Lafi, M. (2019, April). The impact of artificial intelligence on software testing. In 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT) (pp. 565-570). IEEE.