

Research on AI-Empowered Software Test Case Generation

Jin Ping

Computer Science School, Wuhan Donghu College, Wuhan, 430212, China

Keywords

Large Language Models;
Code Generation;
Automated Test Case;
Generation;

ABSTRACT

To resolve the problem that manually writing test cases required substantial effort and to promote software test automation, This paper explores the practice of applying AI technologies combined with large language models to test case generation within the software testing workflow. It leverages large language models to analyze and extract test cases from requirement documents, and constructs an automated test case generation framework by specifying standard input and output values of test samples, test coverage scope, and AI agent roles. Subsequent verification of generated test cases and iterative tuning of prompts and model parameters enable rapid generation of requirement-based test cases, delivering notable improvements in boundary testing and stress testing while shortening the test development cycle. This work expands application scenarios for the automated generation of software test cases and the engineering deployment of large language models.

1.Introduction

Throughout the whole software development lifecycle, software testing is an indispensable part. It acts as a key factor in preserving the reliability and security of software systems. As business processes grow more intricate in contemporary software systems, high coupling between modules and frequent iterative updates have become common realities.

In traditional software testing workflows, test cases are mostly designed manually. This method comes with many obvious drawbacks. It consumes substantial time and incurs high costs. Besides, testers hold different levels of professional experience, which results in uneven quality of generated test cases. Manual test design also makes it difficult to fully cover boundary conditions, abnormal branches and negative test

scenarios. When requirements change rapidly in particular, the workload required to maintain and update test cases surges, and the cost of regression testing keeps rising accordingly.

Most conventional testing techniques depend on scripts written in advance by testers. This kind of automation follows predefined rules. It can only adapt to fixed business workflows and lacks the ability to independently identify hidden abnormal scenarios.

Driven by the rapid advances in machine learning and large language model (LLM) technologies, the automatic generation of test cases powered by artificial intelligence has emerged as a prominent research topic within software development and software quality assurance. The AI-enabled automatic test case generation method integrates cutting-edge AI technologies such as machine learning, deep learning and natural language processing, representing a new testing paradigm. With outstanding semantic analysis capability and advantages in code processing, the large language model (LLM) can intelligently parse requirement documents, identify core functional points, automatically generate test cases and conduct quality assessment. This technical domain leverages static code analysis, search-based test generation, semantic understanding from large models, autonomous model learning, parameter optimization, prompt engineering and other techniques. It automatically extracts business logic from requirement documents, interface definitions, source code and historical test data, then produces structured test cases and executable test scripts.

This paper reviews existing technologies, representative tools and evaluation metrics for AI-based automated test case generation. It summarizes limitations in current research and puts forward future outlooks.

The research aims to realize practical engineering deployment of AI-driven automatic test case generation, fundamentally addressing the pain points of conventional manual test case design. Specifically, its significance lies in the following aspects:

1) Efficiency Improvement: Automated test case generation cuts down manual writing time and shortens the testing cycle, helping enterprises speed up software iteration.

2) Quality Assurance: Automated testing covers more boundary conditions and abnormal scenarios, reduces the risk of missed tests, and improves software quality as well as user experience.

3) Cost Savings: It lessens reliance on senior testers and saves labor costs. Meanwhile, the overall testing cost can be reduced by optimizing testing procedures.

4) Technological Advancement: It promotes the practical application of AI technology in software testing, and offers new solutions for enterprises to achieve intelligent transformation.

2.Literature Review

With the continuous enrichment of software functions and growing complexity of system architectures, user demands have become personalized and diversified. As agile development and DevOps gain widespread adoption, software development prioritizes rapid iteration and efficient delivery, imposing higher requirements on the efficiency and quality of software testing. Conventional manual testing can no longer keep pace with modern software development workflows. Early automated test generation techniques, represented by Search-Based Software Testing (SBST), generate test inputs via genetic algorithms and random search. Popular tools such as EvoSuite and Pyguitar efficiently produce unit test inputs, yet suffer from evident limitations: they cannot accurately interpret natural-language requirement specifications, and the generated test cases exhibit poor readability, making them difficult to deploy in real business scenarios. Software testing serves as a critical gatekeeper for product quality and operational risk mitigation. The rapid industrial evolution has posed substantial practical challenges to traditional testing process.

The emergence of artificial intelligence technologies, especially large language models (LLMs), has opened up the transformation pipeline from natural language requirements to test scripts. These models can interpret unstructured documents and test case samples to generate test cases, making up for the deficiencies of traditional

approaches. In recent years, numerous scholars at home and abroad have carried out research on the automatic generation of AI-powered test cases, achieving phased results in scenarios such as unit testing, interface testing and system testing.

With the rise of large language models, the research paradigm has shifted. Capable of understanding natural language requirements, parsing code syntax and reasoning over business branches, LLMs can convert product requirements and interface documents into structured test cases. Supported by few-shot prompting, retrieval-augmented generation (RAG) and other approaches, the technology lowers the barrier to test case authoring and improves testing efficiency. It has been extensively explored in the fields of interface testing, web-based system testing and unit testing.

2.1 Overseas Research Status

Early research on the automatic generation of AI-based test cases was mostly built upon machine learning and search algorithms. Many scholars have attempted to apply metaheuristic algorithms such as genetic algorithms and particle swarm optimization to the task of test case generation. Taking code coverage as the optimization objective, these algorithms automatically search for appropriate input data to improve the coverage of unit testing. Nevertheless, such methods rely heavily on the structural information of source code and cannot directly read and parse requirement documents written in natural language.

After the emergence of pre-trained language models, the focus of relevant research has gradually shifted toward large code models built on the Transformer architecture. Models such as CodeBERT, CodeT5 and the GPT series have been widely adopted for code comprehension and unit test generation tasks. The proliferation of generative AI is disrupting this polarization model. Tasks previously considered "safe zones," such as abstract analysis, code generation, and text writing, are being drawn into the substitution scope on a massive scale (Eloundou et al., 2023). Celik A and Mahmoud (2026) presents three research questions and further categorizes the findings based on the common themes. Their findings highlight the opportunities and challenges associated with the use of LLM in this domain. Schäfer

et al. (2024) conducted a large-scale empirical evaluation of the effectiveness of LLM for automated unit test generation, requiring no additional training or manual intervention.

From the perspective of practical implementation, most overseas studies focus on the automation of unit testing at the code level. Many researchers also evaluate the effectiveness of model-generated test cases and construct a multi-dimensional evaluation indicator system covering code coverage, test case executability, defect detection capability and other dimensions. However, existing research has identified a notable issue: although test cases produced by large models are usually grammatically correct, they may contain logical errors. When dealing with programs featuring complex business logic and state dependencies, the generation performance of the model degrades significantly.

2.2 Domestic Research Status

Although domestic research started relatively late, it is more application-oriented, with a primary focus on industrial scenarios such as interface testing and business system testing. Several enterprises have carried out explorations on the automatic generation of test cases. Numerous studies combine knowledge graphs with large models. The knowledge graph is used to extract requirement entities, decompose requirements, formulate rules and model relationships for business demands, so as to build a business logic knowledge base. Combined with LLMs, this approach can generate test cases complying with business rules and address the problem that standalone large models tend to violate business constraints.

Studies by domestic scholars concentrate more on adapting to Chinese requirement scenarios. General large models usually show biases in understanding Chinese business requirements and struggle to accurately capture the practical business logic of domestic projects. To tackle this challenge, they mostly adopt domain fine-tuning and prompt engineering optimization to continuously enhance model performance. This enables models to automatically extract test points from PRD documents and requirement specifications and generate corresponding test cases. They also consider various challenges in engineering deployment, such as the

automatic verification of AI-generated test cases, long-term maintenance of mass test cases, and automatic update mechanisms for test cases when requirements change. Nevertheless, most existing solutions remain prototype systems that have only completed preliminary exploration and verification. When deployed to large and complex enterprise-level systems, these approaches will encounter many practical obstacles, including intricate business rules, difficulties in constructing test data, and the fact that hallucinations induced by large models cannot be completely eliminated. Wang G et al(2026). provide insights into the limitations of LLM-based generation, analyzing the reasons for live and uncovered mutants, and the impact of different mutation operators on generation effectiveness.

Compared with theoretical research, the industry pays more attention to the practical deployment effect of technologies. Many existing industrial studies design intelligent testing frameworks for micro service interface scenarios. By automatically parsing various parameters and constraint rules in OpenAPI interface documents, these frameworks can batch-generate ready-to-use interface test cases and automated scripts, which effectively meet the regular interface testing requirements of enterprises.

The rapid iteration of Large Language Model (LLM) technology enables automatic parsing of business logic and extraction of constraint conditions from multi-source materials, such as natural-language requirement documents, product PRDs, API interface definitions, source code, historical defect repositories and existing test cases. The technology can automatically generate structured test cases containing preconditions, test steps, input parameters, expected results, priority levels and test types. Some frameworks can even directly output executable test scripts. Combined with prompt design, learning from input and output requirement documents, parameter tuning and prompt optimization, large models are capable of producing test cases that comply with requirement specifications. These test cases can generally cover boundary scenarios, positive scenarios, and abnormal negative scenarios.

At the same time, academia and industry have gradually recognized that one-shot generation relying solely on large models suffers from hallucinations, logical errors

and other defects. For this reason, closed-loop agent schemes featuring the "generation-execution-feedback-iterative optimization" workflow have become a key research focus in recent years.

3. Theoretical Basis and Research Methods

As an essential part of the software life-cycle, software testing aims to detect potential system defects at an early stage and ensure software robustness. Traditional testing methods such as boundary value analysis, stress testing and equivalence partitioning, black-box and white-box testing are manually. But Manual testing is only suitable for systems with low business complexity. When it comes to large-scale systems, including micro-service architectures, distributed systems and multi-terminal collaborative business systems, the testing workload becomes enormous. Substantial human resources and efforts are required to guarantee the smooth progress of testing activities. The rapid advancement of large language models brings new ideas for the automatic generation of test cases, and exploratory studies have emerged to realize test case automation with AI approaches.

3.1 Theoretical Basis: Current Mainstream Technical Routes

1) Search and Evolutionary Algorithm-based Test Generation

Centered on genetic algorithms and particle swarm optimization, this technical route relies on code control flow graphs and abstract syntax trees. With code coverage set as the optimization objective, it automatically searches for input data. Its advantage lies in strong capability to cover code branches. However, it only targets source code and cannot comprehend business requirements. The generated test cases suffer from poor readability. This method is mostly applied to unit testing and difficult to deploy for business system testing.

Its core idea is to convert test case generation into an optimization problem. Taking code coverage and defect detection probability as the fitness function, it continuously iterates and adjusts input parameters through genetic algorithms, particle swarm optimization and other algorithms to traverse program branch paths. Meanwhile, it is often combined with symbolic execution and dynamic symbolic

execution to collect program path constraints and solve for inputs that satisfy branch conditions.

The strength of this approach is its powerful ability to explore code branches. It can automatically discover code paths easily overlooked by human testers, making it suitable for unit testing. Nevertheless, the method is heavily dependent on source code and unable to understand business semantics. The generated test inputs carry no business meaning and lack clear test steps and expected results. It exhibits low efficiency when dealing with complex business state machines and scenarios involving multi-module interaction.

2) Traditional Machine Learning-based Test Case Filtering and Generation Assistance

This technical route mainly adopts traditional machine learning models such as classifiers and ranking models, which are trained on historical test datasets, historical defect repositories and code metric data. The models can identify high-risk code modules and frequent defect types, prioritize existing test cases and eliminate redundant ones. They can also predict bug-prone scenarios based on historical defect features to assist in generating targeted test cases. Classification and sequence models are used to learn the mapping relationships between historical test cases, requirements and source code, so as to capture patterns of test points from historical data.

Its advantage lies in suitability for regression testing scenarios. It enables intelligent allocation of testing resources by prioritizing the testing of high-risk modules and reducing the workload of regression testing. Compared with search algorithms, it runs faster. However, it relies on high-quality labeled datasets and suffers from weak generalization ability. It struggles to handle brand-new business scenarios. This approach mostly reuses existing testing knowledge and can hardly create completely new business scenarios.

3) Large Language Model-Driven Test Case Generation (Current Mainstream Research Direction)

The large-model approach represents the most widely studied and industrially active technical route in recent years. It can be divided into four categories according to its enhancement schemes:

a) Basic Prompt Engineering-based Approach

This method takes requirement documents, interface specifications, business rules and parameter value ranges as contextual information. Combined with standard input and output samples, it restricts the output format of the model through prompts, enabling large language models to directly generate structured test cases. The approach includes two paradigms: zero-shot and few-shot. The zero-shot paradigm does not require reference examples to be provided to the model, while the few-shot paradigm introduces a small number of high-quality test cases as reference samples to guide the model to generate results following the predefined enterprise test case templates.

b) Retrieval-Augmented Generation (RAG)

RAG builds a vector database from the project's private knowledge base, including product requirements, interface specifications, historical test cases, business constraints and database dictionaries. Before test case generation, relevant business materials are retrieved and fed into the large language model as context. This approach effectively alleviates the knowledge cutoff issue of large models and reduces hallucinations such as fabricated interfaces and business rules, ensuring that the generated test cases conform to the actual project business.

c) Fine-tuning

Fine-tuning adopts high-quality internal enterprise test case datasets to fine-tune the base large model, allowing the model to learn the enterprise's internal test case templates, business terminology and testing styles. The fine-tuned model produces outputs that better comply with business specifications. Nevertheless, this method requires a large volume of high-quality labeled datasets and incurs relatively high fine-tuning costs, making it suitable for large-scale projects with long-term stability.

d) Closed-loop Iterative Generation via Testing Agents

This approach constructs a multi-agent system and performs closed-loop operations including requirement analysis, test case generation, test coverage evaluation, and result verification through task decomposition. After the model automatically generates test cases, the tests are executed repeatedly and execution results are collected. Error reports and coverage metrics are fed back to the large

language model, which automatically revises faulty test cases through continuous tuning. Such iterative optimization continues until the expected final outcomes are achieved.

This solution can greatly reduce manual correction costs, yet it features a sophisticated system architecture and has high reliance on automated execution platforms. The model accepts natural language requirements, source code and interface documents as inputs, and outputs test points, test cases and automated scripts. Its advantages lie in support for unstructured inputs and the capability to generate multiple types of test cases for unit, interface and system testing with relatively low development costs. Its drawbacks include model hallucinations and logical errors in generated test cases. It tends to lose business constraints in long-text scenarios, and the generated test cases are less satisfactory for scenarios with complex data structures.

4) Multimodal Fusion-based Test Generation (Cutting-edge Research Direction)

Knowledge graphs are employed to formalize business rules, symbolic execution is used to verify program paths, and large language models take charge of natural language understanding and text generation. Multiple technologies complement one another. This represents a mainstream direction in current academic research, aiming to mitigate hallucinations and improve business compliance.

Current cutting-edge research has begun to explore multimodal inputs. In addition to textual requirements, the model supports image inputs such as UI prototypes, flowcharts and ER diagrams. It identifies page elements and business flowcharts to automatically generate functional test cases for Web front-ends and APP interfaces. This research direction remains in the laboratory stage with few engineering deployment cases.

3.2 Research Methods

This paper adopts large language models (LLMs) to implement intelligent automatic generation of test cases. By means of requirement information extraction, in-depth document learning and iterative tuning of model output results, this method achieves relatively favorable experimental outcomes. The overall architecture and specific workflow of the proposed method are illustrated as follows.

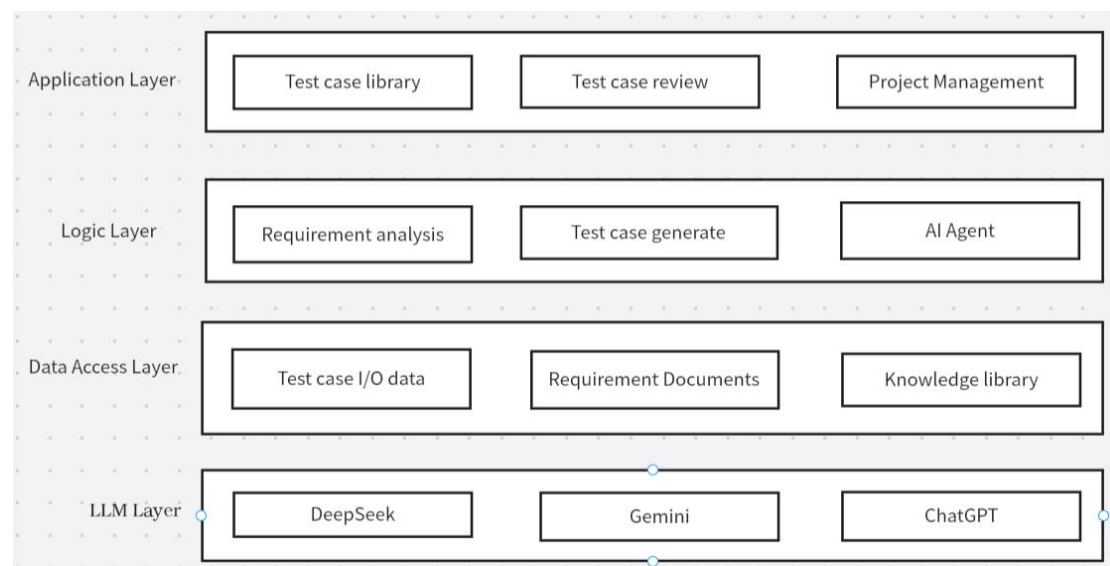


Figure 1 Architecture of AI Test Case Generation System

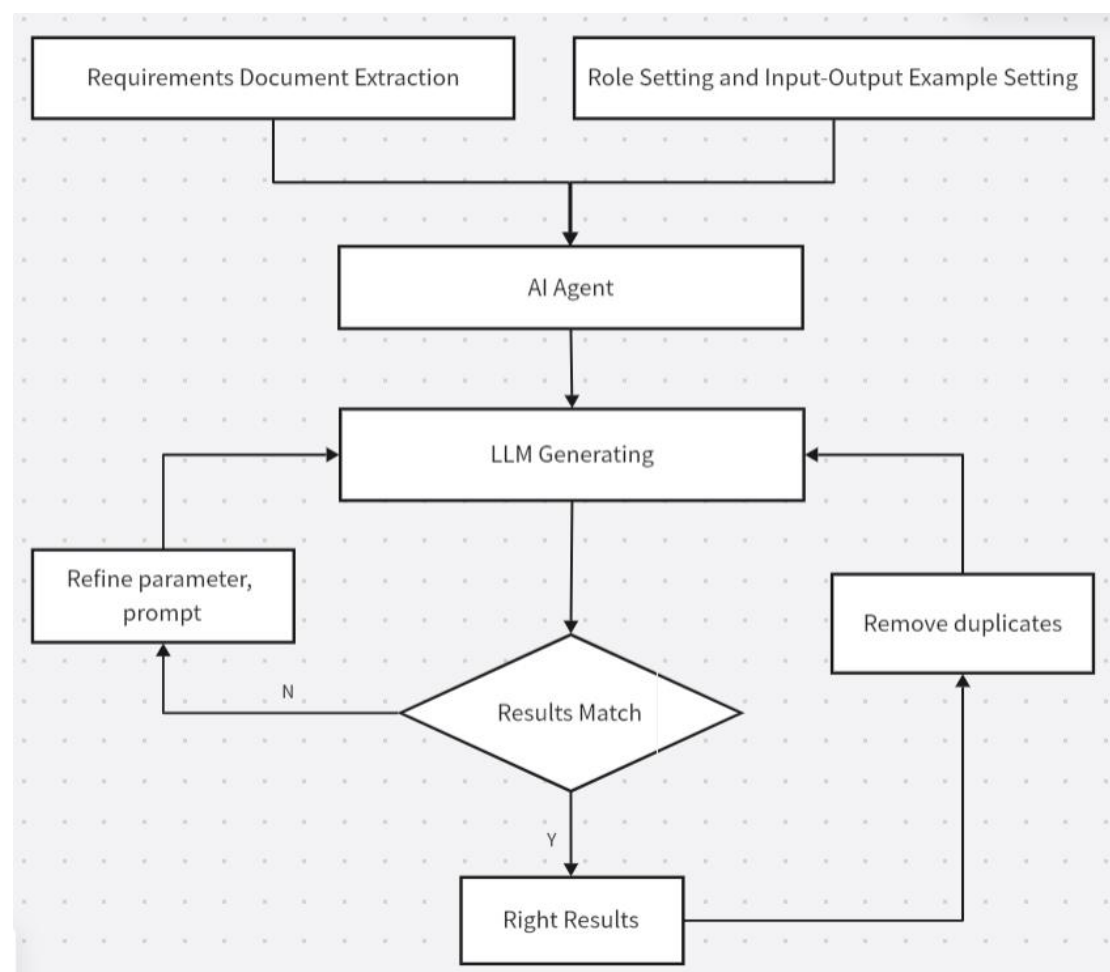


Figure 2 Flowchart of AI Test Case Generation

1) Requirement Parsing and Test Point Extraction

The primary step of test case generation is the decomposition of requirement documents. In this stage, the large language model is guided to split independent sub-requirements from the original requirement documents. Taking sub-requirements as the basic unit, the model learns the specifications of test cases corresponding to inputs and outputs within the documents.

At the technical implementation level, the method performs in-depth parsing of requirement documents to automatically extract test elements including functional points, boundary conditions and abnormal scenarios. Parsed objects contain information such as test inputs, expected outputs, default values, and upper and lower parameter limits. The large language model is leveraged to understand document text, and role instructions are assigned to the model to undertake the task of test case authoring. Input materials for this stage include original requirement documents, expected output samples and customized prompts. By identifying verbs, nouns and inherent logical relationships in requirement texts, the model finally constructs a structured test requirement tree.

Application Scenario: For example, in automotive software requirement documents, the system can automatically identify the functional point of "user button operation", extract boundary conditions such as "detecting active and inactive states of the electric mode" and "parking brake activation and deactivation", and generate corresponding test cases.

2) Diverse Test Case Generation

With deep learning models and large language models (e.g., Transformer architecture, GPT series, ERNIE, Qwen, etc.), multiple types of test cases can be generated to cover different scenarios including functional testing, performance testing and security testing. Reinforcement learning algorithms can be introduced in the research to continuously optimize the diversity and scenario coverage of test cases based on user feedback.

The core implementation idea of this scheme is to use large language models to complete batch generation of test cases from the extracted test requirement points. During generation, various constraints of test cases need to be defined in accordance

with document specifications, including inputs and expected outputs, default parameters, as well as maximum and minimum parameter values. In the dynamic generation process, users can evaluate the validity of test cases to dynamically adjust model parameters and improve the quality of generated test cases. Through manual inspection of test case results, users can optimize prompts repeatedly and train the model to search for optimal solutions.

Application Scenario: When generating functional test cases, the system can automatically consider various situations including normal flows, abnormal flows and boundary values to ensure test comprehensiveness.

3) Code Logic Analysis

It focuses on dedicated verification of boundary scenarios and abnormal test cases within code logic. This method leverages Abstract Syntax Trees (AST) to parse code structures, identify key code fragments such as loops and conditional branches, and generate corresponding test data accordingly.

Application Scenario: For instance, when analyzing the code for abnormal tire pressure alarm detection, the system can automatically generate test cases covering multiple scenarios including high pressure, low pressure and extreme values to guarantee code robustness.

4) Intelligent Management of Test Cases

a) Knowledge Base Construction

Testers can add previously validated test cases for specific scenarios into the knowledge base as a knowledge inventory. Test cases stored in the knowledge base can be reused for similar subsequent problems. Similar to a knowledge caching system, it can rapidly retrieve solutions for identical problems.

Application Scenario: For example, in an automotive software system, when a new function of "window closure check" is added, the system can recommend historical test cases for "window closure". Users only need minor adjustments for reuse. In such cases, users can generate test cases more rapidly.

b) Visual Interface

The system provides a visual Web interface and a standalone desktop client, enabling testers to edit, execute, track and evaluate the effectiveness of

model-generated test cases. The front-end interface integrates a visualization module for test reports, which can display core metrics such as test coverage and defect distribution in real time. The web version is developed based on the Vue framework, while the back-end adopts the Spring Boot framework to ensure efficient data interaction and visualization. The executable desktop program is implemented using Python.

Application Scenario: Testers can intuitively view the execution status of test cases through the interface, quickly locate problems and improve testing efficiency.

3.3 Quality Evaluation of Automatically Generated Test Cases

For the quality evaluation of AI automatically generated test cases, to achieve comprehensive and multi-level standardized assessment and avoid the one-sidedness of a single evaluation dimension, this paper divides the evaluation system into two core categories: quantitative metrics at the code level, and combined qualitative and quantitative metrics at the business level. The code-level evaluation centers on objective quantitative data, focusing on the underlying technical performance, code adaptability and test effectiveness of test cases. It intuitively assesses the technical quality and implementation efficiency of AI-generated test cases through quantifiable and precise data, serving as the core basis for measuring the basic reliability of automated test cases.

Table 1 Quality Evaluation of Test Cases at Code Level

Dimension	Key Evaluation Metrics and Description
Efficiency	Generation time consumption, labor saving ratio, maintenance cost upon requirement changes
Quality	Test case execute ability, defect detection capability, test case redundancy, logical correctness
Coverage	Statement coverage, branch coverage, requirement coverage, functional point coverage

Table 2 Quality Evaluation of Test Cases at Business Level

Dimension	Key Evaluation Metrics and Description
Syntax and Format Compliance	Whether test cases conform to enterprise templates with complete fields (preconditions, steps, expected results)
Business Logic Correctness	Consistency with requirement documents and absence of errors violating business rules
Scenario Coverage	Coverage quantity of positive scenarios, boundary values, abnormal scenarios and negative scenarios
Readability	Clear description ,Comprehensible
Executability	Generated parameters and interface are authentic and valid for direct execution with valid content
Redundancy	No massive duplicate or highly similar invalid test cases

There are notable challenges in the current evaluation process. Code coverage can be calculated automatically, whereas the correctness of business logic requires manual inspection. Automated assessment methods for verifying the correctness of test cases are lacking. Large language models exhibit inherent randomness in outputs; given identical inputs, the generated results may vary across multiple runs, which further increases the difficulty of quality evaluation. Model hallucination stands out as the most prominent issue in quality assessment. The model may fabricate non-existent interfaces, parameters and business constraints, producing seemingly reasonable yet actually unexecutable and meaningless test cases.

4. Challenges In AI-enpowered Test cases generated

4.1 The Problem of Model Hallucination

Model hallucination represents the most uncertain factor when using LLM to generate test cases. In complex business scenarios involving long texts, ambiguous

semantics and intricate business logic, large language models are limited in business logical reasoning. They struggle to fully comprehend requirements and may even misunderstand them, which consequently leads to missing branches, logical errors and invalid test cases.

For large distributed business systems with multi-role permissions, state transitions and multi-system interactions, requirement documents are often unstructured and ambiguous. It is difficult for large language models to automatically identify implicit business constraints in documents, and hidden rules are prone to be overlooked. Generating high-quality test cases for time-sequence dependencies, concurrent scenarios and abnormal fault injection scenarios remains highly challenging.

Potential solutions include establishing joint laboratories with university. Cooperation and conduct collaborative technical research, and inviting industry experts to guide model optimization.

4.2 Lack of a Unified Quality Evaluation System

At present, no universally recognized and standardized evaluation system for AI-generated test cases has been established in academia and industry. Code coverage can only measure performance at the code level and cannot assess business correctness; coverage does not equal defect detection capability. Many studies only measure coverage and lack evaluation of defect detection effectiveness. Most indicators at the business level rely on manual scoring, which carries strong subjectivity and is difficult for large-scale automated evaluation. How to construct a comprehensive evaluation system that takes into account code coverage, business compliance and defect detection capability remains a research challenge.

4.3 Data Security Risks

When private requirements, interfaces and business data from projects are fed into large models, there is a certain risk of business information leakage. Many highly regulated industries necessitate private deployment and bring additional operation and maintenance costs. In addition, the input code and business requirements may contain sensitive information, creating risks of data leakage during model training and inference. Encryption technologies and anonymization

processing can be adopted, together with regular security audits. The training datasets of base large models contain a large amount of open-source code and public test documents, which introduces potential copyright risks. The liability definition for AI-generated test cases remains unclear. If erroneous AI-generated test cases result in missed tests and trigger online failures, there are no unified specifications to determine liability attribution.

4.4 Knowledge Base Maintenance Cost

The RAG approach requires continuous maintenance of the vector knowledge base. When requirements change or interfaces iterate, the knowledge base needs to be updated synchronously; otherwise, outdated test cases will be generated. In scenarios with frequent requirement changes, it is difficult for AI to automatically identify change points and batch-update existing test cases, and manual confirmation is still required in many cases.

5. Research Findings

This study evaluated the capability of LLMs including ChatGPT and Gemini to automatically generate test cases from requirement descriptions. The experiment sampled inputs from the use case library, invoked each model to produce test cases with input and output data, then performed manual review the deduplicated results and quality.

The findings show that LLM can rapidly generate basic functional test cases and reduce manual writing effort. But LLM tend to miss implicit boundary conditions and edge cases. A portion of generated test cases are logically inconsistent or redundant, which necessitates manual review after result deduplication before the final review results are confirmed.

Prompt design significantly impacts output quality. Well-structured prompts help improve the consistency of generated test cases. Nevertheless, fully autonomous end-to-end test case generation without human oversight remains unreliable for high-risk business scenarios.

6. Future trends

6.1 Future Technical Directions

Future AI-based automatic test case generation technology will evolve from merely "generating textual test cases" toward a full-link, intelligent, human-machine collaborative quality engineering system. The main research directions are as follows:

1) **Integration of Business Knowledge Graph and Large Language Models** Construct a business knowledge graph to structurally model business entities, state transitions and constraint rules. Integrate the knowledge graph with LLMs to enhance reasoning capability for complex business scenarios, mitigate model hallucinations, accurately identify implicit business constraints, and improve the quality of test cases generated for complex business scenarios.

2) **Closed-loop Automated System of Testing Agents** Continuously improve the closed-loop agent framework of "requirement parsing → test case generation → script generation → automatic execution → result collection → defect localization → automatic test case repair". Test cases are iteratively optimized continuously based on feedback from test execution to reduce manual intervention. The framework enables automatic updates of test suites under continuous integration environments and adapts to agile development and DevOps pipelines.

3) **Construction of a Standardized and Automated Comprehensive Quality Evaluation System** Research automated business verification metrics. Combined with static business rule verification, dynamic execution results and mutation testing, a comprehensive scoring model capable of automatic calculation will be constructed to replace extensive manual reviews and realize automated quality scoring for AI-generated test cases.

4) **Domain-specific Models and Private Deployment for Vertical Industries** For highly compliant industries such as finance, electric power, healthcare and industrial software, build industry-specific test knowledge bases and conduct domain fine-tuning. Adopt private on-premises deployment schemes to ensure that business data does not leave the intranet and meet the requirements of industrial data security and compliance audit.

5) **Deepening of Multi-modal Test Generation Technology** Integrate graphic and textual information such as prototype diagrams, business flow charts, sequence

diagrams and ER diagrams to realize the direct generation of front-end and interface test cases from UI prototypes and business flow charts. This reduces reliance on pure-text requirement documents and adapts to the prototype-driven product development model. AI undertakes a large number of repetitive tasks: batch generation of positive, boundary and basic abnormal test cases, automatic test script generation and regression test execution. Testers can focus on higher-value work including business risk modeling, design of complex and security scenarios, review of AI-generated content, formulation of quality strategies, and root cause analysis of defects. A human-machine collaborative testing paradigm is formed, where AI delivers outputs in batches while humans control risk.

6.2 Conclusion

The AI-based automatic test case generation technology has evolved from search-algorithm-driven unit test input generation to LLM-driven business-level test case generation. With technologies such as LLM, RAG and testing agents, AI can greatly reduce the human effort required for test case design and improve the coverage efficiency of boundary and abnormal scenarios. It accommodates the engineering demands of agile development and has demonstrated certain practical value in the fields of interface testing and unit testing.

In the long run, the adoption of AI is not intended to completely replace test engineers. Instead, it improves their work efficiency, reshapes the working paradigm of software testing, and builds a human-machine collaborative testing system. As technologies including knowledge graphs, multi-modal large models and testing agents mature further in the future, AI will be deeply integrated into the entire DevOps pipeline to realize a closed loop for the automatic generation, execution and maintenance of test cases. It will continuously enhance software testing efficiency and software delivery quality. AI-empowered testing will usher in a new era.

References

- Rastenis M, Chou B, Just R. Automated Software Test Generation at Industry Scale Using a Multi-Agent Architecture and Workflow Integration[C]//2026 IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (SEIP). ACM, 2026. <https://doi.org/10.1145/3786583.3786918>
- Andrzejewski M,Dubicka N,Podolak J,et al.Automated Test Generation Using Large Language Models[J].Data,2025,10(10):156. <https://doi.org/10.3390/data10100156>
- Hasan N B,Islam M A,Khan J Y,et al.Automatic High-Level Test Case Generation using Large Language Models[EB/OL].(2025-03-23)[2025-12-23]. <https://arxiv.org/abs/2503.17998>
- Lu Qingmei, Li Junli, Gao Yuan. Exploration of university computer general education based on AI[J]. Frontiers in Educational Research,2024,7(12):80-85. <https://doi.org/10.25236/FER.2024.071212>
- Esposito G, Sanchez M, Fratini F, et al. Embedding AI ethics in technical training:A multi-stakeholder pilot module emphasizing co-design and interdisciplinary collaboration at Rome Technopole[J]. Education Sciences,2025,15(10):1416. <https://doi.org/10.3390/educsci15101416>
- Eloundou,T.,Manning,S.,Mishkin,P.,&Rock,D.(2023).GPTs are GPTs:Anearly look at the labor market impact potential of large language models.arXiv preprint arXiv:2303.10130. <https://arxiv.org/abs/2303.10130>
- Thanachawengsakul N, Suwathanpornkul I, Charoensuk O. Confirmatory factor analysis of the computational thinking learning competency measurement model of students in the bachelor of education program in computer at rajabhat

- university[J]. International Education Studies,2024,17(5):123-134.
<https://doi.org/10.5539/ies.v17n5p123>
- Iznaga Y S, Rato L, León J L. Integrating Large Language Models into Automated Software Testing[J]. Future Internet, 2025,17(10):476.
<https://doi.org/10.3390/fi17100476>
- Tufano M,Deng S K,Sundaresan N,et al.Methods2Test:A data set of focal methods mapped to test cases[C]//Proceedings of the 19th International Conference on Mining Software Repositories.2022:30-41.
<https://doi.org/10.1145/3524842.3528009>
- Nan Z, Guo Z, Liu K, et al. Test Intention Guided LLM-Based Unit Test Generation[C]//2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). IEEE, 2025:1026- 1038.
<https://doi.org/10.1109/ICSE55347.2025.00243>
- Wang G, Xu Q, Briand L, et al. Mutation-Guided Unit Test Generation With a Large Language Model[J]. IEEE Transactions on Software Engineering, 2026,52(5):1657- 1671. <https://doi.org/10.1109/TSE.2026.3682975>
- Omri S,Al-Dahoud A,A IDahoud S.Deep learning for software defect prediction:A survey[J].Applied Sciences,2020,10(12):4328
- RIEDMAIER S, PONN T, LUDWIG D, et al. Survey on scenario-based safety assessment of automated vehicles[J]. IEEE Access, 2020, 8: 87456-87477.
- SCHULDT F, RESCHKA A, MAURER M. A method for an efficient, systematic test case generation for advanced driver assistance systems in virtual environments[J]. Automotive Systems Engineering II, 2018: 147-175.
- Biagiola M, Ghislotti G, Tonella P. Improving the Readability of Automatically Generated Tests using Large Language Models[C]//2025 IEEE/ACM 47th

International Conference on Software Engineering.IEEE,2025:1412- 1423.
<https://doi.org/10.1109/ICSE55347.2025.00110>

Andrzejewski M, Dubicka N, Podolak J, et al. Automated Test Generation Using Large Language Models[J]. Data, 2025,10(10):156.
<https://doi.org/10.3390/data10100156>

Schäfer M, Nadi S, Eghbali A, et al. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation[J]. IEEE Transactions on Software Engineering, 2024,50(1):85- 105.
<https://doi.org/10.1109/TSE.2023.3334955>

Escalante-Viteri A, Mauricio D. Artificial Intelligence in Software Testing: A Systematic Review of a Decade of Evolution and Taxonomy[J]. Algorithms, 2025,18(11):717. <https://doi.org/10.3390/a18110717>

Celik A, Mahmoud Q H. A Review of Large Language Models for Automated Test Case Generation[J]. Machine Learning and Knowledge Extraction, 2025,7(3):97. <https://doi.org/10.3390/make7030097>

Amalfitano D, Faralli S, Hauck J C R, et al. Artificial intelligence applied to software testing: A tertiary study[J]. ACM Computing Surveys, 2023, 56(3): 1-38. <https://doi.org/10.1145/3616372>