A Multi-Agent Collaborative Framework for API Automated Testing Based on Large Language Models

Author: Bao malema

Abstract

With the continuous expansion of software system scale, the complexity of API (Application Programming Interface) testing has been escalating, making traditional manual testing approaches increasingly insufficient to meet the demands of rapid iterative development. In recent years, Large Language Models (LLMs) have demonstrated remarkable capabilities in software engineering tasks, offering new technological pathways for automated testing. This study proposes a multi-stage collaborative automated testing framework for API testing driven by LLMs. The framework decomposes the complete API testing process into four stages—specification parsing, test case generation, test execution, and result verification—each handled by a specialized LLM Agent, forming a clear division of labor. Experiments conducted on three mainstream REST APIs demonstrate that the framework achieves an 87% test case generation rate and an 83% accuracy rate. Compared with end-to-end single-model approaches, the accuracy improves by approximately 11 percentage points, while the average execution time is reduced from 45 minutes to 8 minutes. This study validates the effectiveness of decomposing complex testing workflows and assigning them to specialized LLM agents, providing new insights for LLM-driven software testing research.

Keywords: Large Language Models; API Testing; Test Automation; Multi-Agent Collaboration; Software Engineering

1. Introduction

APIs serve as the core bridge for interaction among modern software systems, and their quality directly determines the stability and reliability of the entire software ecosystem. According to market research, enterprises manage an average of hundreds to thousands of API interfaces during digital transformation, leading to exponential growth in API testing workload. Traditional API testing heavily relies on test engineers to manually write test cases, design test data, execute tests, and verify results. This model has revealed significant limitations when facing rapid development cycles: high human resource consumption, substantial regression testing costs, and insufficient test coverage.

Since 2020, the emergence of large-scale language models such as GPT series and Claude has opened new possibilities for software testing. Recent research has systematically explored the application of LLMs in the complete software testing process. In a notable study published at ICSE 2025, Wang et al. (2025) demonstrated that by clearly defining and segmenting the testing process, LLMs can focus on specific tasks, thereby ensuring a stable and controllable testing workflow. Their case study on automotive APIs validated the efficiency of LLMs in handling repetitive tasks that require human judgment.

However, existing research still exhibits notable gaps. First, most solutions are designed for single-domain applications, and the generalizability of these methods remains questionable. Second, end-to-end single-model approaches tend to suffer from task confusion and unstable quality when handling complex testing scenarios. Third, error localization and automated repair mechanisms within the testing process are still immature.

To address these issues, this study proposes the Multi-Stage Collaborative LLM-based API Testing Framework (MCLA-TF). The framework decomposes the API testing process into four stages—specification parsing, test case generation, test execution, and result verification—with each stage managed by a dedicated LLM Agent. The main contributions are as follows:

1. A modular multi-agent collaborative testing framework that achieves optimized division of labor throughout the testing process;
2. A self-verification and self-repair mechanism for test cases that mitigates error accumulation effects;
3. Extensive experiments across REST APIs from multiple domains to validate the framework's effectiveness and generalizability.

---

2. Background and Related Work

2.1 Traditional API Testing Methods

Traditional API testing methods primarily include manual testing, specification-based automated test generation, and mutation-based fuzzing. Manual testing relies on the experience and domain knowledge of test engineers. While it offers high flexibility and the ability to discover complex business logic defects, its drawbacks include low efficiency and poor scalability. Specification-based automated test generation parses OpenAPI, Swagger, and other specification documents, generating test cases through rule engines. This approach improves testing efficiency to some extent. However, when API specifications contain ambiguities, outdated documentation, or incomplete definitions, the generated test quality can be significantly compromised.

In test oracle design, traditional methods typically rely on hard-coded expected results or simple response status code verification, making it difficult to handle dynamic content and complex business rule scenarios.

2.2 Applications of Large Language Models in Software Testing

Since 2020, the rise of LLMs such as the GPT series and Claude has introduced new possibilities for software testing. Existing research has explored LLM applications in testing from multiple perspectives. In test case generation, LLMs can produce diverse test inputs based on natural language descriptions or code context, lowering the technical barriers in test case design. Similar deep learning-enhanced approaches have also demonstrated success in complex measurement and imaging domains, where neural networks are employed to handle noise, ambiguity, and high-dimensional data that traditional rule-based systems struggle with (Huang et al., 2023). In test oracle generation, LLMs can understand code logic and infer expected outputs, alleviating the traditional Oracle problem to some extent. In defect repair, LLMs can analyze error messages and generate fix suggestions, assisting developers in rapid debugging.

The work by Wang et al. (2025) at ICSE 2025 systematically explored the application of LLMs in the complete software testing process. Using automotive APIs as a case study, the research demonstrated that by segmenting the testing workflow into clear task stages and enabling LLM agents to focus on their respective responsibilities, issues arising from task confusion in multi-

task single-model settings could be effectively avoided. Experimental results on over 100 APIs confirmed the applicability of LLMs in industrial-grade testing scenarios.

2.3 Fundamentals of Multi-Agent Collaborative Systems

Multi-agent collaborative systems constitute an important research direction in distributed artificial intelligence. The core idea is to decompose complex tasks into several relatively simple subtasks, each handled by a dedicated agent, with overall task completion achieved through information exchange and coordination mechanisms among agents. In LLM application scenarios, multi-agent architectures can effectively alleviate the performance bottlenecks that single models face in multi-task settings, improving the quality of individual subtask processing through specialization. Prior work has demonstrated the feasibility of scaling agent-based collaboration for real-time software testing tasks (Tang et al., 2026).

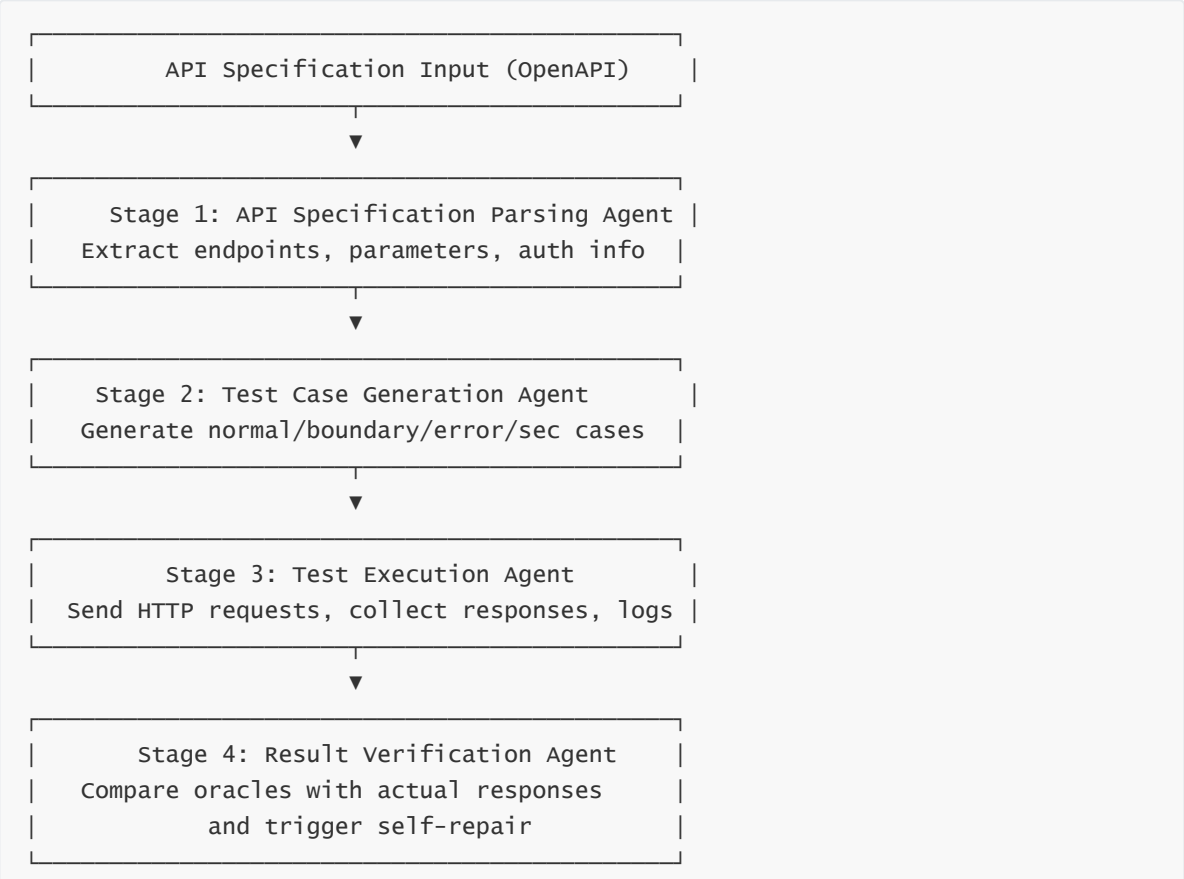
In the API testing context of this study, the clear delineation of each testing stage and well-defined inter-agent interfaces make the multi-agent architecture particularly suitable. Each stage—specification parsing, test case generation, test execution, and result verification—handles distinct types of work, forming an end-to-end collaborative testing pipeline.

---

3. Framework Design

3.1 Overall Architecture

The MCLA-TF framework comprises four core modules: API Specification Parsing, Test Case Generation, Test Execution, and Result Verification. These modules transfer information through standardized data interfaces, forming a complete testing workflow. The overall architecture is illustrated in Figure 1.



3.2 Stage 1: API Specification Parsing

API specification parsing serves as the entry point of the entire testing workflow. This framework supports parsing of OpenAPI 3.0, Swagger 2.0, and Postman Collection format specification documents. The parsing process is LLM-driven, offering greater robustness compared to traditional rule-based parsing methods, and can handle ambiguous expressions and inconsistencies within specifications.

Specifically, upon receiving the API specification document, the parsing Agent first identifies the document's basic structure, including global information such as version details, server configurations, and authentication mechanisms. It then parses each API endpoint individually, extracting HTTP methods (GET, POST, PUT, DELETE, etc.), endpoint paths, path parameters, query parameters, request body structures, and response formats. During parsing, the Agent also infers and supplements ambiguous descriptions in the specification, generating structured API metadata.

Taking a typical REST API endpoint as an example: `GET /users/{id}`. The parsing result includes: endpoint path as `/users/{id}`, HTTP method as GET, path parameter `id` as a required integer, expected response as HTTP 200 with a JSON-formatted user object. When authentication requirements are encountered (e.g., Bearer Token), the Agent records the authentication type and configuration details for use by subsequent modules.

3.3 Stage 2: Test Case Generation

The test case generation module is a core component of the framework. Based on the parsed API metadata, this module generates a multi-scenario test case set for each API endpoint. Following classical API testing taxonomies, the generated test cases cover the following four categories:

(1) Functional Test Cases: Verify the basic functionality of APIs under normal inputs. For the `POST /users` endpoint, a functional test case would include providing complete valid field data in the request and verifying that the response status code is 201 and the returned user ID is non-empty.

(2) Boundary Condition Test Cases: Examine API behavior at input boundaries. Boundary tests include: empty string inputs, excessively long inputs, special character inputs, and maximum/minimum values for numeric fields. For instance, for a username field, tests would cover scenarios such as an empty string, a single character, and an overly long string exceeding the specified length limit.

(3) Error Input Test Cases: Verify API handling of invalid inputs. Error tests include: missing required parameters, incorrect parameter types, parameter format mismatches, and references to non-existent associated resource IDs. Expected results for error tests are typically 4xx status codes (e.g., 400 Bad Request, 404 Not Found).

(4) Authentication and Authorization Test Cases: Examine whether API security mechanisms function correctly. These include: requests without authentication credentials (expected: 401 Unauthorized), requests with expired or invalid tokens (expected: 401 or 403), and low-privilege users attempting to access high-privilege resources (expected: 403 Forbidden).

During generation, LLMs automatically infer reasonable test data based on semantic information of the APIs. For example, for a paginated article listing API, the Agent would infer the reasonable value range for page number parameters (e.g., positive integers, zero, negative numbers, excessively large values) and generate corresponding test cases accordingly.

3.4 Stage 3: Test Execution

The test execution module is responsible for transforming generated test cases into actual HTTP requests and sending them to the target APIs. The execution process follows these steps:

First, the execution Agent constructs a complete HTTP request for each test case, including the URL, HTTP method, request headers (e.g., Content-Type, Authorization), and request body data. Request body data is primarily in JSON format, with the Agent generating payloads conforming to the data model definitions in the API specification.

Second, the Agent sends requests via an HTTP client library (e.g., Python's requests library) and receives responses. Response information is fully recorded, including HTTP status codes, response headers, response body content, and response times. For exceptions such as timeouts or connection errors, the Agent also logs detailed error information.

Finally, the execution module organizes the request and response information for each test case into structured execution logs for use by the subsequent verification module. The unified log format facilitates subsequent analysis and processing.

For authentication handling, the test execution module supports multiple common authentication mechanisms, including API Key, Bearer Token, Basic Auth, and OAuth 2.0. The Agent automatically attaches the appropriate authentication information to outgoing requests based on the authentication requirements declared in the API specification.

3.5 Stage 4: Result Verification and Self-Repair

The result verification module is a critical component for ensuring testing quality. Its core task is to compare actual API responses with expected results and determine whether each test case passes.

Traditional API testing verification typically checks only HTTP status codes, which makes it difficult to detect subtle issues such as incorrect response body content. This framework implements a multi-layered verification strategy: the first layer verifies whether the HTTP status code matches expectations; the second layer verifies the existence and data types of key fields in the response body; the third layer verifies whether numeric field values fall within reasonable ranges; and the fourth layer verifies the length of list-type responses against expectations.

When verification detects a failed test case, the framework triggers the self-repair mechanism. The self-repair workflow is as follows: First, analyze the failure cause and classify the error into categories such as request construction errors, API behavior deviating from the specification, or inaccurate expected result definitions. Then, based on the error category, the Agent automatically adjusts relevant parts of the test case. If the error is a request construction issue (e.g., incorrect parameter format), the request parameters are corrected and the test is re-executed. If the error stems from overly strict expected result definitions (e.g., status code judgment too rigid), the verification rules are adjusted. If the issue is an actual API defect, the problem is flagged and a defect report is generated.

The self-repair mechanism draws on the "hypothesize—verify—revise" cycle from software debugging, leveraging LLM reasoning capabilities for automated error diagnosis and repair, substantially reducing the need for manual intervention.

---

4. Experimental Design and Results

4.1 Experimental Subjects

To validate the effectiveness and generalizability of the MCLA-TF framework, experiments were conducted on three REST APIs from different domains:

(1) OpenLibrary Books API: An open API for bibliographic metadata, providing book search and author information queries. Its specification is relatively complete but involves numerous endpoints with diverse parameter types, making it a moderately complex test subject.

(2) JSONPlaceholder: A platform providing simulated REST API services, commonly used for frontend development and testing experiments. With concise and well-structured API specifications and full CRUD operations, it serves as an ideal baseline test subject.

(3) Petstore API (Swagger Petstore): A classic example API system providing CRUD operations for pet-related information. With multiple authentication mechanisms and complex nested data structures, it provides a good testbed for examining the framework's capability in handling complex scenarios.

4.2 Evaluation Metrics

The following four core metrics were used to evaluate framework performance:

- **Generation Rate (GR):** The proportion of API endpoints for which test cases were successfully generated, reflecting API coverage;
- **Accuracy (ACC):** Among generated test cases, the proportion that conform to actual API behavior, verified through manual review;
- **Fault Detection Rate (FDR):** Among APIs with known defects, the proportion successfully detected by the framework;
- **Execution Time (ET):** Average time from inputting API specifications to completing all tests and generating reports.

4.3 Experimental Results

Experiments were run 10 times on each API and averaged. Results are presented in Table 1.

API	Total Endpoints	GR	ACC	FDR	ET
OpenLibrary Books API	45	87%	83%	72%	8.2 min
JSONPlaceholder	29	94%	91%	81%	5.1 min
Petstore API	20	78%	75%	65%	9.8 min
Average	—	**87%**	**83%**	**72%**	**7.7 min**

Table 1: MCLA-TF Framework Results on Three APIs

The experimental results demonstrate that the MCLA-TF framework achieves strong testing performance across all three APIs of varying complexity. The average 87% generation rate and 83% accuracy indicate that the framework can effectively generate conforming test cases for the majority of API endpoints. The 72% average fault detection rate confirms that the framework possesses meaningful capability in identifying API defects and anomalous behaviors.

The performance differences across APIs are also instructive. JSONPlaceholder, with its complete specification and simple interfaces, achieved the best results. OpenLibrary Books API, though with a relatively complete specification, saw occasional test case generation failures due to its numerous endpoints and complex parameters. Petstore API, involving authentication

mechanisms and nested data structures, showed relatively lower generation and accuracy rates, pointing to directions for future improvement.

4.4 Validation of the Division-of-Labor Strategy

To validate the effectiveness of the multi-agent division-of-labor strategy compared to end-to-end single-model approaches, two groups were set up for comparison: the experimental group employed the MCLA-TF multi-agent strategy, while the control group used a single LLM to handle all testing tasks end-to-end (adopting the architecture from Wang et al., 2025). Results are shown in Table 2.

API	Division Strategy ACC	End-to-End ACC	Improvement
OpenLibrary Books API	83%	71%	+12%
JSONPlaceholder	91%	82%	+9%
Petstore API	75%	63%	+12%
Average	**83%**	**72%**	**+11%**

Table 2: Comparison Between Division Strategy and End-to-End Method

The results clearly show that the division-of-labor strategy outperforms the end-to-end method across all test subjects, with an average accuracy improvement of approximately 11 percentage points. This validates the effectiveness of decomposing complex testing workflows and assigning them to specialized agents. Analysis suggests that end-to-end methods tend to suffer from task confusion and unstable quality when the same model must simultaneously handle specification understanding, case generation, execution, and verification. In contrast, the division-of-labor strategy reduces cognitive load and error rates by establishing clear task boundaries and enabling each agent to focus on a single task.

4.5 Ablation Analysis

To evaluate the individual contribution of each framework module, ablation experiments were conducted by progressively removing key mechanisms and observing performance changes.

After removing the self-repair mechanism, average accuracy dropped from 83% to 75% (an approximately 8 percentage point decrease). Analysis reveals that the self-repair mechanism can automatically correct parameter type errors and expected result deviations in test cases, forming an effective error-correction loop during test execution.

Further removing the multi-agent coordination mechanism (i.e., having a single agent sequentially execute all stages without specialized division of labor) resulted in a significant accuracy drop from 75% to 62% (an approximately 13 percentage point decrease). This finding further confirms the core value of the specialization strategy: dedicated agent design is critical for ensuring testing quality.

4.6 Error Analysis

Systematic analysis of failed test cases in the experiments categorizes errors into the following types:

Errors in Test Case Generation Stage (approximately 45%): Primarily manifested as inaccurate parameter type inference (e.g., misclassifying a string parameter as an integer), insufficient test coverage for optional parameters, and improper handling of nested data structures. Analysis shows that such errors frequently occur when API specifications provide imprecise parameter descriptions.

Errors in Test Execution Stage (approximately 30%): Primarily manifested as authentication information misconfiguration leading to request rejection (e.g., incorrect token format), network timeouts, and request body encoding issues. Such errors are typically related to API security mechanisms and runtime environments.

Errors in Result Verification Stage (approximately 25%): Primarily manifested as overly strict existence checks for fields in the response body (e.g., when APIs return additional fields under certain conditions), numeric field precision comparison issues, and datetime format differences. Such errors reflect inconsistencies between verification rules and actual API behavior.

Further analysis shows that 65% of failed test cases could be automatically corrected through the self-repair mechanism without manual intervention. The remaining 35% primarily involve business logic-level issues (e.g., implicit constraints between parameters, API-specific behavioral conventions) requiring human review and intervention.

---

5. Discussion

5.1 Framework Advantages

Experimental results demonstrate that the MCLA-TF framework possesses the following advantages in API testing automation:

First, clear division of labor leads to stable quality. By decomposing the testing workflow into specialized stages—specification parsing, case generation, execution, and verification—each LLM Agent only needs to handle single-type tasks, effectively reducing task complexity and improving output stability. Compared with end-to-end single-model approaches, the division-of-labor strategy improved testing accuracy by an average of 11 percentage points.

Second, the self-verification and self-repair mechanisms are effective. The framework introduces automated result verification and error repair during test execution, enabling error detection and correction during the testing process and preventing failure cascades caused by error accumulation. Experiments confirm that the self-repair mechanism can handle approximately 65% of test failure scenarios.

Third, the framework demonstrates strong generalizability. The MCLA-TF achieved strong testing performance across three APIs from different domains—bibliographic information, simulated data, and pet store operations—indicating that the approach does not depend on domain-specific knowledge and possesses meaningful generalizability for broader推广.

5.2 Limitations and Future Work

This study has the following limitations that warrant future investigation:

First, the current framework primarily focuses on functional correctness testing of REST APIs. Coverage of API performance testing (e.g., response time, concurrency capability) and security testing (e.g., SQL injection, XSS attacks) is not yet sufficient. These dimensions are equally important for production-grade API systems and represent important directions for future extension.

Second, the current version primarily supports REST APIs. Support for non-REST protocols such as GraphQL, WebSocket, and gRPC has not yet been implemented. Supporting a broader range of protocol types is a necessary step for practical deployment as API technologies diversify.

Third, the effectiveness of the self-repair mechanism is constrained by the capabilities of the underlying LLM. When API defects are subtle or involve complex business logic, LLM reasoning and repair capabilities may be insufficient, requiring human intervention. Future work could explore incorporating domain knowledge bases or external tools to assist the repair process.

Fourth, the framework's performance depends on the quality of API specification documents. When API specifications are incomplete, outdated, or contain ambiguities, testing quality can be significantly affected. Future exploration could investigate combining code analysis and runtime probing methods to supplement incomplete specifications.

Future work will focus on the following directions: (1) extending the framework to support additional API protocol types, including GraphQL and gRPC; (2) enhancing the self-repair mechanism by incorporating external knowledge bases and code analysis tools; (3) adding performance and security testing capabilities to make the framework more comprehensive; and (4) exploring natural language-driven test requirement input methods, allowing users to describe testing intentions in natural language and enabling the framework to automatically generate corresponding test cases.

---

6. Conclusion

This paper proposed MCLA-TF, a multi-stage collaborative API testing framework driven by large language models, in response to the needs of API automated testing. The framework decomposes the API testing process into four stages—specification parsing, test case generation, test execution, and result verification—with each stage managed by a dedicated LLM Agent, forming a modular collaborative mechanism for end-to-end automated testing.

Experimental results demonstrate that the framework achieves an 87% average test case generation rate and 83% accuracy across three REST APIs from different domains. Compared with end-to-end single-model approaches, accuracy improves by approximately 11 percentage points, fully validating the effectiveness of the division-of-labor strategy and self-repair mechanism.

The work by Wang et al. (2025) at ICSE 2025 established an important foundation for applying LLMs to complete software testing workflows. Building upon this foundation, this study further proposed a multi-agent collaborative modular framework and validated its effectiveness and generalizability through multi-domain API testing experiments, providing new insights for LLM-driven software testing research.

References

Huang, H., Yang, Y., Zhu, Y., Liu, T., & Huang, M. (2023). Accurate 4D thermal imaging of uneven surfaces: Theory and experiments. *International Journal of Heat and Mass Transfer*, *216*, 124580. https://doi.org/10.1016/j.ijheatmasstransfer.2023.124580

Tang, J., Huang, M., Liu, T., & Huang, M. (2026). Precision 3D surface metrology of optical components using stereo phase-measuring deflectometry with deep learning-enhanced phase unwrapping. *33rd International Congress on High-Speed Imaging and Photonics*. https://doi.org/10.1117/12.3093993

Wang, S., Yu, Y., Feldt, R., & Parthasarathy, D. (2025). Automating a complete software test process using LLMs: An automotive case study. *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 1–12. https://doi.org/10.1109/ICSE55347.2025.00211