From Requirements to Executable Tests: LLM-Based System Test Generation for REST APIs

Jaroslav Kochanovskis, Asta Slotkienė

Vilnius University, Faculty of Mathematics and Informatics,
Universiteto g. 3, Vilnius, Lithuania
jaroslav.kochanovskis@mif.stud.vu.lt

Abstract. Modern software systems increasingly rely on APIs, making efficient testing essential for ensuring reliability and correctness. However, creating system tests from natural language requirements is typically a manual and time-consuming process. This research proposes an approach for automatically generating executable REST API tests from natural language requirements using large language models (LLMs) and OpenAPI specifications. A prototype system was developed and evaluated on two real-world APIs. The results show that the approach can generate executable pytest tests, achieve full endpoint reachability coverage, and provide stable automated test execution.

Keywords: software testing, automated test generation, natural language processing, large language models, system testing, REST API.

1 Introduction

Software testing is an essential part of the software development lifecycle, ensuring that systems function correctly, meet user requirements, and remain reliable in different situations. As software systems become more complex and are used in critical domains such as healthcare, transportation, and finance, the importance of effective testing continues to increase. Without proper testing, software failures may lead to significant financial, operational, or safety consequences [1].

Traditionally, test cases are created manually by analysing system requirements and designing testing scenarios. This process requires significant effort from testers and developers and is often time consuming and prone to human error. As a result, manual testing may lead to incomplete test coverage or inconsistencies between requirements and implemented tests. These challenges become more significant in modern development

DOI: https://doi.org/10.15388/LMITT.2026.13

environments such as Agile and DevOps, where rapid development cycles require faster testing processes [2].

Recent advances in artificial intelligence and natural language processing have created new opportunities for automating software testing activities. Several studies have explored the use of LLMs for generating test cases or assisting in software testing tasks [3, 4, 5, 6, 7]. However, many existing approaches rely on analysing source code or generate test descriptions that still require manual implementation.

This research proposes an approach for automatic generation of system test cases from natural language specified requirements. The proposed method applies natural language processing and LLMs to analyse requirement descriptions and transform them into executable tests. The goal of this research is to improve requirement coverage and reduce the manual effort required for test creation.

2 Research Methodology

This research follows a design experiment methodology. First, a prototype system was developed to automatically generate executable REST API tests from natural language requirements and OpenAPI specifications. The effectiveness of the proposed approach was then evaluated experimentally by generating and executing tests for selected APIs and analysing the obtained results.

The proposed method transforms natural language functional requirements into executable system level tests for REST APIs. The overall research workflow consists of several stages: preparation of the requirement dataset, loading of the API specification, automatic test generation using a LLM, validation of generated test code, and execution of the tests against the target API.

The complete workflow of the proposed approach is illustrated in Figure 1. The workflow begins with the selection of the target API and preparation of the requirement dataset. The system then loads the OpenAPI specification and initializes the test generation environment. For each requirement, a generation context is constructed using the requirement text, the API specification, and additional configuration parameters such as the API base URL. A LLM is then used to generate candidate pytest test code. The generated code is automatically validated to ensure syntactic correctness and compliance with predefined structural rules. If validation

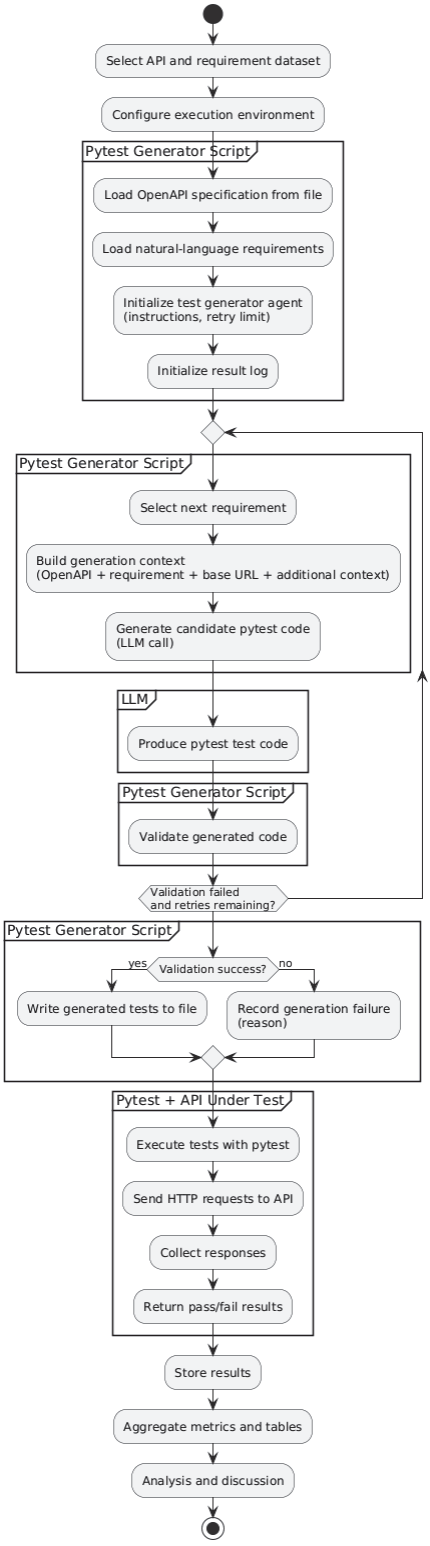

Figure 1. UML Activity diagram for process approach.

fails, the generation step is repeated until the retry limit is reached. Valid test cases are saved as executable pytest files and later executed during the experimental evaluation phase. Finally, the execution results are collected and aggregated for further analysis.

The approach relies on two primary input artifacts: natural language requirements and the OpenAPI specification. A requirement describes the expected behaviour of the system from the user's perspective. In this research, requirements are written in natural language and represent functional scenarios, authorization rules, error handling cases, or boundary conditions. An example requirement is: "When a client provides a valid access key, the system shall return the list of supported currency symbols.". Using natural language requirements reflects common industry documentation practices and allows evaluation of the approach under realistic conditions.

The OpenAPI specification provides a structured description of the API under test. It defines available endpoints, supported HTTP methods, request parameters, authentication mechanisms, and expected responses. In the proposed approach, the specification is used to provide contextual information during test generation and to reduce ambiguity when interpreting requirements.

A prototype implementation of the proposed approach was developed in Python using an agent-based interaction with a LLM. The architecture of the prototype is presented in Figure 2.

The prototype consists of several main components responsible for orchestrating test generation and validation. The main controller initializes the generation process, loads the OpenAPI specification, and prepares the test generation request. A test generation agent interacts with a LLM to produce candidate pytest test code based on the provided requirements and API context. Generated code is then passed to a validation module that checks the syntactic correctness and structure of the produced tests before they are accepted by the system.

For each natural language requirement $r \in R$, the system constructs a structured generation request that combines the requirement text, the OpenAPI specification S, the API base URL U, and additional execution context C. The request object is represented using typed data models implemented with the Pydantic library. This ensures that the input information passed to the generation agent follows a consistent schema and reduces ambiguity when constructing prompts for the language model.

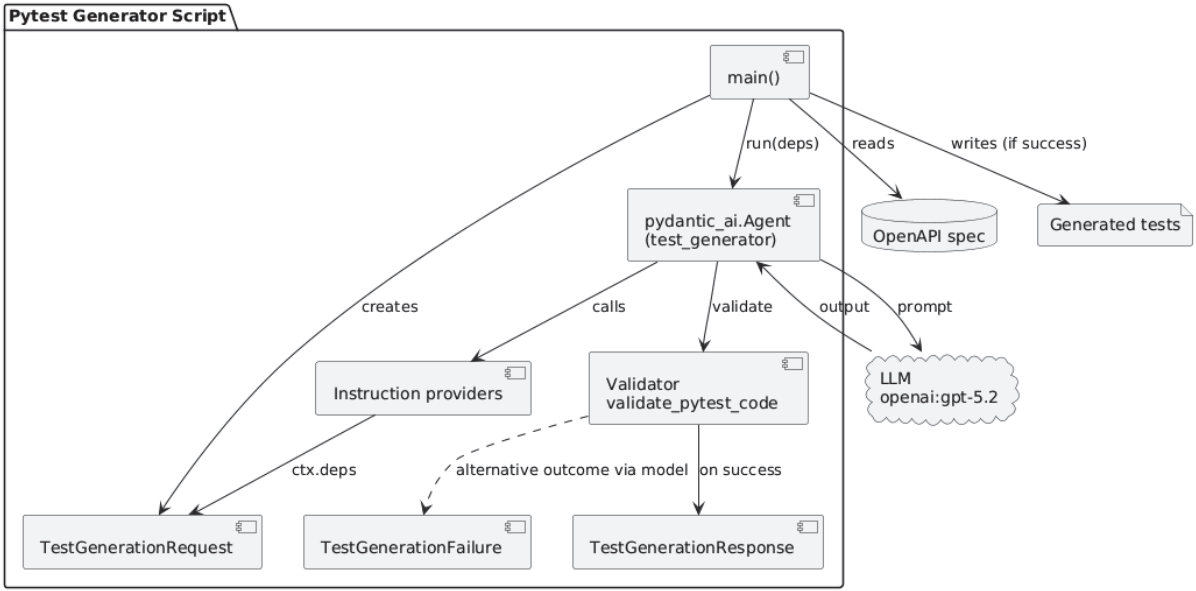

Figure 2. UML Component diagram of implemented prototype.

The generation agent uses this structured input to produce candidate pytest test code. The generated output is not accepted directly; instead, it is processed through a validation pipeline that verifies several structural and syntactic constraints. The validation stage ensures that the generated code contains at least one pytest test function, includes the required libraries for HTTP request execution and test handling, and represents syntactically valid Python code. These checks help ensure that the produced tests can be executed without manual modification. This validation stage acts as a safeguard against incomplete or malformed outputs produced by the language model and ensures that only structurally correct and executable tests are accepted by the system.

If the generated output does not satisfy these constraints, the generation process is repeated until a predefined retry limit is reached. This mechanism allows the system to automatically recover from invalid or incomplete generation attempts. Successfully validated tests are saved as executable pytest test files.

The generation agent was implemented using the PydanticAI framework, which enables structured interaction with LLMs through typed input and output schemas. The OpenAI GPT-5.2 model, accessed through the OpenAI API via the PydanticAI framework, was used for test generation due to its strong performance in code generation and natural language understanding tasks. The model is guided by a fixed instruction prompt that defines the role of the model as a pytest test generation expert. The instructions specify that the model must transform natural language requirements into complete executable pytest test cases, including required imports, HTTP request execution using the requests library, and assertions validating expected API responses.

The generated test files are executed separately using the pytest framework during the experimental evaluation phase. Execution results, including pass and fail status and runtime errors, are then collected and analyzed to assess the effectiveness of the proposed approach.

3 Experiment and Results

The proposed approach was evaluated using two REST APIs defined by their OpenAPI specifications: Fixer API and PayPal Checkout Orders API. Since production APIs typically do not provide natural language requirements directly, the requirement datasets were prepared manually from the

OpenAPI specifications with the assistance of a LLM. The requirements describe expected system behaviour, authorization constraints, error handling scenarios, and boundary or state dependent conditions.

The prototype implementation was developed in Python. For each requirement, the system automatically generated executable pytest test cases, validated them, and executed them against the target API without manual modification. The experiment followed a fixed procedure consisting of requirement selection, loading the OpenAPI specification, automatic test generation using a LLM, validation of generated tests, execution using the pytest framework, and collection of execution results. The evaluation focused on metrics suitable for black box API testing, including requirement coverage, endpoint coverage, test executability, execution stability, and efficiency. Code level metrics such as code coverage were not applicable because the internal implementation of the tested systems was not accessible.

The evaluation dataset consisted of 67 natural language requirements derived from the OpenAPI specifications of the evaluated APIs. The Fixer API dataset contained 37 requirements, while the PayPal Checkout Orders API dataset contained 30 requirements. These requirements describe expected system behaviour such as data retrieval, authorization handling, error conditions, and state dependent operations. The requirements were written in natural language to reflect realistic software documentation practices and to evaluate the ability of the proposed approach to interpret informal requirement descriptions.

For each requirement in the dataset, the system generated one or more pytest test cases targeting relevant API endpoints and scenarios. Generated tests included both positive cases and negative scenarios such as invalid parameters, missing authorization, or incorrect input values. Table 1 summarizes the overall results of the generated test suites for both evaluated APIs.

Table 1. Test generation and execution results.

API	Require-ments	Generated tests	Average tests per require-ment	Passed tests	Failed tests
Fixer API	37	189	5.1	146	43
PayPal Orders API	30	102	3.4	91	11

For the Fixer API, the system generated 189 test cases from 37 requirements, resulting in an average of 5.1 tests per requirement. Among these tests, 146 passed successfully while 43 failed during execution. For the PayPal Checkout Orders API, the system generated 102 test cases from 30 requirements, with an average of 3.4 tests per requirement. Out of these tests, 91 passed successfully and 11 failed. In both cases, the generated tests exercised all endpoints defined in the OpenAPI specifications, achieving full endpoint reachability coverage.

Each generated test case is associated with the requirement from which it was derived. During the test generation process, requirements are processed individually, and the resulting pytest test files are stored together with identifiers referencing the originating requirement. This one-to-many mapping between requirements and generated tests establishes traceability between natural language specifications and the produced test artifacts. As a result, it is possible to determine which requirements are covered by the generated tests and to evaluate requirement coverage based on the existence of at least one executable test case per requirement.

In addition to endpoint coverage, requirement coverage was evaluated to determine whether each natural language requirement resulted in at least one generated test case. Requirement coverage is defined as the proportion of requirements for which the system successfully generated executable tests and is calculated as:

$$Requirement\ Coverage = \frac{R_{covered}}{R_{total}} \times 100\%,$$

where $R_{covered}$ represents the number of requirements with at least one generated test and R_{total} represents the total number of requirements in the dataset. A requirement is considered covered if at least one generated test corresponding to that requirement can be executed successfully, even if other generated tests for the same requirement fail during execution. For example, if a requirement produces multiple test cases and only a subset of them passes, the requirement is still considered covered. This definition reflects the goal of evaluating the ability of the proposed approach to transform natural language requirements into executable test artifacts. Using this definition, the proposed approach achieved 100% requirement coverage (67 of 67 requirements), indicating that each requirement in the dataset was successfully transformed into at least one executable test case.

Although some generated tests failed during execution, these failures were mainly caused by external API constraints rather than incorrect test generation. Examples include undocumented API behaviour, subscription plan limitations, authorization restrictions, or differences between the OpenAPI specification and the actual runtime behaviour of the system.

The evaluated APIs also differ in their endpoint structures and supported HTTP methods. The Fixer API consists exclusively of GET endpoints that provide read only access to currency data. In contrast, the PayPal Checkout Orders API includes multiple HTTP methods, including GET, POST, PATCH, and DELETE, representing more complex resource management operations such as order creation, updates, and tracking. To analyse how the generated tests performed across different endpoint types, the execution results were grouped by HTTP method classification. Figure 3 presents the pass and fail ratio of generated tests grouped by endpoint class.

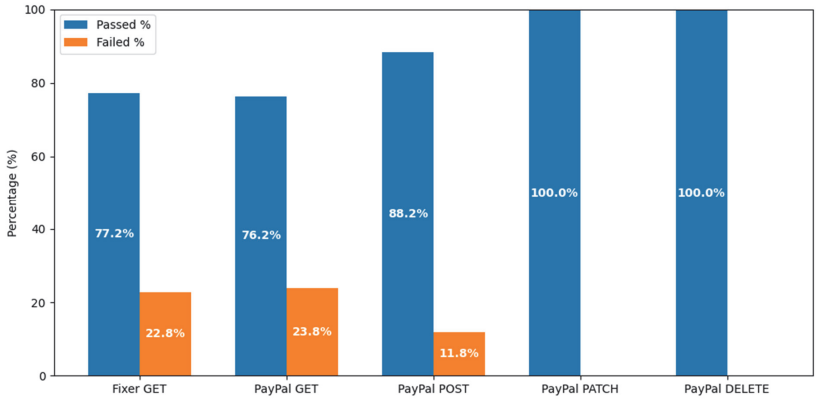

Figure 3. Pass/Fail Ratio by Endpoint Class.

The results show that tests targeting PATCH and DELETE endpoints achieved full execution success, while most failures occurred in GET and POST operations. These failures were primarily related to authorization constraints, state dependent behaviour, and inconsistencies between API documentation and runtime responses. The classification analysis demonstrates that the proposed approach can generate executable tests for a variety of REST interaction types, including retrieval, creation, modification, and deletion operations.

Execution stability was evaluated by repeatedly executing the generated test suites under unchanged conditions. Each test suite was executed five times for both APIs. Across repeated runs, test outcomes remained consistent. Tests that passed or failed did so deterministically in every execution, and no cases were observed where a test alternated between passing and failing across runs. These results indicate that the generated tests themselves are stable and deterministic, and that observed failures are reproducible and attributable to external system constraints.

The efficiency of the proposed approach was evaluated by measuring the time required to automatically generate, validate, and execute system level tests for each API. For the Fixer API, the average execution time was approximately 27.89 seconds, while for the PayPal Checkout Orders API the average execution time was approximately 32.84 seconds. The slightly higher execution time for the PayPal API reflects its greater complexity, including multi step workflows and stricter authorization requirements. Despite these differences, the overall execution time remained within a similar range for both APIs, indicating that the proposed approach scales reasonably with increasing API complexity.

Overall, the experimental results demonstrate that the proposed approach can automatically generate executable system level tests from natural language requirements and OpenAPI specifications. The generated tests successfully exercised the entire API surface of the evaluated systems while maintaining stable execution behaviour across repeated runs. Although some generated tests failed during execution, these failures were primarily caused by external API constraints or specification inconsistencies rather than limitations of the test generation process. These findings suggest that LLMs can effectively support automated generation of executable API tests while reducing the manual effort required for test creation.

4 Comparison with Related Work

Recent studies have explored the use of natural language processing and LLMs for automated test generation. For example, CiRA [8] focuses on extracting logical conditions from natural language requirements to generate acceptance tests that assist human testers. However, the generated outputs are abstract test descriptions rather than executable tests.

Other approaches, such as APITestGenie [10], use LLMs to generate API test scripts from requirements and OpenAPI specifications. While these methods demonstrate the feasibility of LLM assisted test generation, their evaluation typically focuses on generation success rather than systematic execution analysis or runtime validation.

Table 2. Extended comparison with related work.

Aspect	Proposed	CiRA [8]	APITestGenie [9]	Prompt Engineering Paper [10]
Input type	NL reqs + OpenAPI	NL reqs	NL reqs + OpenAPI	OpenAPI + Prompts
Test level	System level	Acceptance	System level	System level / scenario
Executable output	Yes	No	Yes	No / semi executable
Human intervention	None	Required	Recommended	Recommended
Failure diagnostics	High	Medium	Medium	Not evaluated

In contrast, the proposed approach focuses on generating fully executable system level tests and evaluating them through direct execution against real-world APIs. The experiments include analysis of endpoint coverage, execution stability, and failure diagnostics, providing a more complete evaluation of the generated tests in realistic testing environments.

Table 2 summarizes the main differences between the proposed approach and related work.

5 Threats to Validity

During the experiments, some generated tests failed due to external API constraints such as authorization restrictions, subscription limitations, or inconsistencies between the OpenAPI specification and the actual runtime behaviour of the API. These factors were outside the control of the proposed approach and may influence the observed pass–fail ratios.

The evaluation was performed using two real-world REST APIs with different endpoint structures and authentication mechanisms. Although these APIs represent realistic testing environments, evaluating additional APIs

with different architectures and documentation quality would be necessary to further confirm the general applicability of the proposed approach.

In this work, requirement coverage and endpoint reachability coverage were used to evaluate the effectiveness of the generated tests. However, these metrics primarily measure test generation capability and executability rather than the semantic correctness of the tests. Although passing tests indicate successful execution, they do not necessarily guarantee that the tests fully validate the intended requirement semantics. Techniques such as mutation testing or expert review could be used in future work to further assess the fault detection capability of the generated tests.

6 Conclusions

This paper investigated the feasibility of generating executable system level REST API tests from natural language requirements using LLMs and OpenAPI specifications. The proposed approach enables automated generation of pytest test cases for black box testing scenarios where internal system details are not available. The approach demonstrates how structured generation requests, OpenAPI context, and automated validation mechanisms can be combined with LLMs to support reliable generation of executable system level tests.

Experimental evaluation on two real-world APIs demonstrated that the approach can reliably generate syntactically valid and executable test cases without manual intervention. Although these APIs provide realistic testing environments with different endpoint structures and authentication mechanisms, evaluating only two systems represents a limitation of the current study. Future work will extend the evaluation to a broader range of APIs to further assess the generalizability of the proposed approach. All requirements were successfully transformed into corresponding test artifacts, achieving full requirement–test traceability at the generation level. The generated tests exercised all API endpoints, achieving 100% endpoint reachability coverage for both evaluated systems.

Some execution failures were observed during testing, mainly due to external constraints such as authorization restrictions, subscription limitations, and inconsistencies between API specifications and runtime behavior. However, repeated executions showed stable and deterministic results, indicating that the generated test logic itself is reliable.

Overall, the results show that LLMs can effectively support automated generation of executable API tests from natural language requirements. Future work will focus on improving support for state dependent workflows and integrating the proposed approach into continuous integration environments.

References

[1] X. Jia. The Role and Importance of Soſtware Testing in Soſtware Quality Management. Journal of Industry and Engineering Management. 2023. Available from: https:// doi.org/10.62517/jiem.202303406.

[2] E. Daka, G. Fraser. A Survey on Unit Testing Practices and Problems. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering. 2014, pp. 201–211. Available from: https://doi.org/10.1109/ISSRE.2014.11.

[3] M. Schäfer, S. Nadi, A. Eghbali, F. Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering. 2024, pp. 85–105.

[4] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.G. Lou, W. Chen. CodeT: Code Generation with Generated Tests. In: The Eleventh International Conference on Learning Representations. 2023.

[5] H. Kirinuki, H. Tanno. ChatGPT and Human Synergy in BlackBox Testing: A Comparative Analysis. 2024. Available also from: https://arxiv.org/abs/2401.13924.

[6] J. Fischbach, J. Frattini, A. Vogelsang, D. Mendez, et al. Automatic creation of acceptance tests by extracting conditionals from requirements: NLP approach and case study. Journal of Systems and Software. 2023. ISSN 01641212.

[7] J. W. Lim, T. K. Chiew, M. T. Su, S. Ong, H. Subramaniam, M. B. Mustafa, Y. K. Chiam. Test case information extraction from requirements specifications using NLPbased unified boilerplate approach. Journal of Systems and Software. 2024, volume 211. ISSN 0164-1212.

[8] J. Fischbach, J. Frattini, A. Vogelsang. CiRA: A Tool for the Automatic Detection of Causal Relationships in Requirements Artifacts. 2021

[9] A. Pereira, B. Lima, J. P. Faria. APITestGenie: Automated API Test Generation through Generative AI. 2024. Available also from: https://arxiv.org/abs/2409.03838.

[10] P. T. T. Kyaw, A. Luangsodsai, P. Bhattarakosol. Open API Testing Using Large Language Models and Prompt Engineering. 2025 22nd International Joint Conference on Computer Science and Software Engineering (JCSSE). 2025, pp. 193–198.