VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
PROGRAMME OF STUDY IN SOFTWARE ENGINEERING

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

## Nuo reikalavimų iki vykdomų testų: LLM pagrindu grindžiamas REST API sisteminių testų generavimas

Master's thesis project / Master's thesis

Author: Jaroslav Kochanovskis

Supervisor: Asta Slotkienė

Reviewer: Vaidas Jusevičius

Vilnius – 2026

# Santrauka

Šiuolaikinės programinės įrangos sistemos vis dažniau remiasi REST API sąsajomis, todėl efektyvus API testavimas tampa svarbia programinės įrangos kokybės užtikrinimo dalimi. Tačiau vykdomų sisteminių testų kūrimas iš natūralios kalbos reikalavimų dažniausiai yra rankinis ir daug laiko reikalaujantis procesas. Esami metodai dažnai generuoja tik abstrakčius testų aprašymus arba reikalauja papildomo rankinio darbo.

Šiame darbe siūlomas vykdomų REST API sisteminių testų automatinio generavimo metodas iš natūralios kalbos reikalavimų naudojant didžiuosius kalbos modelius ir OpenAPI specifikacijas. Buvo sukurta prototipinė sistema naudojant Python, PydanticAI karkasą ir OpenAI GPT-5.2 modelį. Siūlomas metodas apjungia reikalavimų analizę, OpenAPI pagrindu formuojamą kontekstą, automatinį validavimą ir testų vykdymą į vieningą procesą.

Metodas buvo įvertintas naudojant Fixer API ir PayPal Checkout Orders API sistemas. Eksperimentiniai rezultatai parodė, kad sistema sėkmingai generuoja vykdomus pytest testus, pasiekia pilną reikalavimų bei galinių taškų pasiekiamumo padengimą ir užtikrina stabilų testų vykdymą. Papildomai atliktas semantinio korektiškumo vertinimas parodė, kad sugeneruoti testai daugeliu atvejų gerai atitiko pradinius reikalavimus.

Rezultatai parodė, kad didieji kalbos modeliai gali efektyviai padėti automatiškai generuoti vykdomus REST API sisteminius testus bei sumažinti pasikartojantį rankinį testų kūrimo darbą.

**Raktiniai žodžiai: programinės įrangos testavimas, automatinis testų generavimas, natūralios kalbos apdorojimas, didieji kalbos modeliai, sistemos testavimas, REST API**

# Summary

Modern software systems increasingly rely on REST APIs, making effective API testing an important part of software quality assurance. However, creating executable system tests from natural language requirements is typically a manual and time consuming process. Existing approaches often generate abstract test descriptions or require additional manual implementation.

This thesis proposes an approach for automatic generation of executable REST API system tests from natural language requirements using large language models and OpenAPI specifications. A prototype system was developed using Python, PydanticAI, and the OpenAI GPT-5.2 model. The proposed approach combines requirement analysis, OpenAPI based context generation, automated validation, and runtime execution into a unified workflow.

The approach was evaluated using the Fixer API and the PayPal Checkout Orders API. Experimental results showed that the system successfully generated executable pytest tests, achieved full requirement and endpoint reachability coverage, and demonstrated stable execution behaviour. In addition, semantic correctness evaluation showed that the generated tests were generally well aligned with the original requirements.

The results demonstrate that large language models can effectively support automatic generation of executable REST API system tests while reducing repetitive manual testing effort.



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

# Contents

# Introduction

Testing is an integral part of software development and is important to ensure applications work correctly, meet user needs, and maintain reliability in different situations. It is a defensive activity disclosing defects and inconsistencies before the release of the software to the end users. This only makes testing more crucial since the failure of these modern software systems grows increasingly complex and might bring devastating consequences in other vital domains for example healthcare, transportation, and finance [Jia23]. Without proper testing, the chances of releasing unreliable or unsafe software increase, and such software can have severe consequences.

Traditionally, test development has been a resource-intensive manual task. Testers and developers have to meticulously read the requirements, think about relevant scenarios, and script test cases to ensure the system behaves as expected. This process is laborious in nature and prone to human error, which can easily lead to test coverage holes or errors in the test process [DF14]. As the complexity of software systems and their requirements increases, these limitations become a serious bottleneck, especially in fast and agile development methodologies like Agile and DevOps.

In recent years, advancement in automation technologies has marked a great revolution in the very domain of software testing. While automated test execution has been widely adopted to optimize repetitive tasks like regression testing, automation in test generation is a relatively young field with great potential. Techniques from artificial intelligence (AI) and natural language processing (NLP) now make it possible to extract meaningful information from unstructured natural language specifications and transform that information into structured, executable tests [LF22; STS+23]. That made major changes, transforming classical manual test creation to intelligent automated processes with less effort, high accuracy, and better scalability. The transition from manual to automated test generation is one of the major steps in modern software engineering methodologies. Automation of test generation does not only speed up the testing process but also improves conformance between requirements and testing by directly deriving test cases from the ultimate source of information: the requirements themselves [MCS+24]. This approach helps bridge the gap between non-technical stakeholders and technical teams, ensuring that delivered software meets its intended goals without introducing unnecessary delays or inefficiencies.

In the past few years, there appeared developed tools, methods and studies regarding automatic tests generation from natural language. One of such tools is TESTPILOT that was created in a research by M. Schäfer et al.[SNE+24]. It could generate unit tests for JavaScript APIs by utilizing contextual information such as function signatures, documentation comments, usage examples, and source code. The key innovation of TESTPILOT is that it does not require additional training or a curated corpus of existing tests, which is a common limitation in other approaches. Instead, it relies on careful prompt engineering to guide the LLM in generating relevant tests. The research demonstrates that TESTPILOT achieves state-of-the-art statement coverage and generates tests with non-trivial assertions[SNE+24]. Other researchers Bei Chen et al.[CZN+23] presented a method called CODET, which focuses on the automatic generation of code solutions and corresponding test cases from programming problem descriptions. The methodology involves leveraging pre-trained language models to generate code snippets based on a given context, which

includes natural language descriptions and existing code snippets. CODET also generates test cases for these code solutions and employs a dual execution agreement to select the best solution based on the consistency of outputs against the generated test cases and agreement with other code samples[CZN+23]. Another researchers Hiroyuki Kirinuki and Haruto Tanno proposed [KT24] investigation of the role of large language models (LLMs), with a focus on ChatGPT, in the context of software testing, mainly black-box testing. This study analyzes ChatGPT's potential in generating test cases based on application specifications and compare the generated cases with ones designed by human testers. The research focuses on the pragmatic use of AI generated test cases, their ability to reveal previously unseen testing perspectives, and how human testers can adopt ChatGPT in their processes in order to enhance testing methodologies[KT24]. Another research by Zongjie Li et al.[LWL+23] suggest a framework called CCTEST designed to test and enhance code completion systems that utilize large language models. The research highlights the issues with current LLM-based code completion systems, particularly their tendency to produce inconsistent outputs. CCTEST employs novel mutation strategies, specifically program structure-consistent mutations, to generate input prompts that can reveal defects in the code completion outputs. The framework also includes a mechanism for repairing these outputs by selecting those that are closest to an "average" appearance of multiple outputs, thereby improving the accuracy of the code completion systems by notable margins[LWL+23]. Another tool that we can find is DROIDAGENT, by Juyeon Yoon et al.[YFY23], is an autonomous Android GUI testing technique leveraging large language models to generate and execute app-specific testing tasks. It sets itself apart by autonomously defining and pursuing testing goals with long-term plans, generating natural language task descriptions that align with developers preference for readable test cases, and focusing on higher-level semantic testing objectives[YFY23]. In other research by Jannik Fischbach et al. [FFV+23] authors presents a method leveraging natural language processing to automate the generation of acceptance tests from natural language requirements. It identifies the manual, error-prone nature of current processes and the challenges posed by informal requirements. The authors focus on conditional statements in functional requirements as a key element that can be systematically extracted to create test cases, addressing limitations in existing approaches that struggle with informal language, grammatical errors, and diverse vocabulary. Next research study, by Jin Wei Lim et al. [LCS+24], explores the automatic generation of test cases from software requirements written in natural language using natural language processing techniques. It aims to extract key test case components (actors, conditions, steps, system responses) from both positive and negative requirements, addressing challenges like ambiguity and inconsistency in NL. The study introduces a unified boilerplate combining elements of Rupp's and EARS frameworks to provide a structured approach for analyzing requirements. The methodology involves pre-processing requirements, extracting test case information, and generating test case documents, leveraging NLP tools like NLTK for improved quality and accuracy.

Comparing each of researches we can see some pattern and possible gaps. Most of it creates tests via analyzing existing code in a system, rather than full product as is [CZN+23; LWL+23; SNE+24]. Other method [YFY23] relied on generating natural language test task for a developers.

It can be noted that the following researches [FFV+23; KT24; LCS+24] are more concentrated on a system as is, but they are also lacking of full automation, since it relies on creating test cases in natural language, so later testers or anyone else could use it for their needs. So technically in end result there is manual work to be done in testing. Ideally, it would be convenient to combine the best of both approaches: the convenience of running tests automatically, like unit tests, and the clarity of aligning with requirements, as seen in natural language test cases.

**The aim of this thesis** is to propose an approach for the automated system test cases generation from natural language requirements, improving requirement coverage.

**Objectives for this work**:

1. Analyse the specifics of natural language features and possibilities for test case generation using NLP.
2. Analyse and compare researches in transforming natural language requirements into formal test cases.
3. Design approach for system test case generation using NLP techniques.
4. Study experiments with the proposed model.
5. Suggest conclusions and recommendations based on experimental results and analysis.

**Expected outcomes**:

1. Proposed and analysed approach for automated system test generation.
2. Created framework or prototype implementation to demonstrate the proposed approach.
3. Conducted comparative analysis to evaluate improvements over existing approaches in terms of requirement coverage.
4. Identified limitations and provided insights into potential enhancements for NLP-based system test generation.
5. Recommended practical applications and future research directions for NLP-driven testing methodologies.

Dissemination of Research Results

Based on the results of this research, a scientific publication was prepared for the national conference "Lithuanian MSc Research in Informatics and ICT". The following article was published in the conference proceedings:

Jaroslav Kochanovskis, Asta Slotkienė. From Requirements to Executable Tests: LLM-Based System Test Generation for REST APIs. Lithuanian MSc Research in Informatics and ICT: Conference Proceedings, May 6, 2026, Vilnius: Vilnius University Press. eISSN 2783-784X. 2026, pp. 118–130. Vilnius University Open Series, eISSN 2783-784X. DOI: `https://doi.org/10.15388/LMITT.2026.13`

The results of this research were also presented as a poster presentation at the national conference "Lithuanian MSc Research in Informatics and ICT", held on May 6, 2026, in Vilnius.

# 1. Literature Review

## 1.1. Foundations of Software Testing

Definition of software testing is the systematic process of evaluating, inspecting, and verifying software systems to detect and correct defects introduced during development or maintenance to ensure that the software meets user requirements and specification standards [Jia23]. This process goal is to validate the correctness, reliability, and efficiency of the software product.

### 1.1.1. Importance of Software Testing

There are multiple reasons of importance of software testing in software engineering. First, it is fundamental part of software quality assurance, ensuring that products are reliable, functions as expected, and safe before deployment. Without testing, software companies risk releasing faulty systems, which can result in financial losses, compromised security, or lost user trust [WHC+24]. Testing also provides significantly to cost reduction, because identifying and fixing defects early in the development lifecycle is less expensive than discovering faults after release. Additionally, testing supports continuous improvement in software development processes, because it gives additional feedback during requirements analysis, design, coding, and integration phases [Jia23].

So software testing is important and when system grows it requires defined standards. In parallel, emerging technologies like large language models are enhancing software testing tasks like creation of test cases or bug detection and that shows, that testing is not only a standard activity but also a continuously evolving process adapting to new technologies [WHC+24].

### 1.1.2. Software testing techniques

Mainly software testing is divided into two categories: black-box testing and white-box testing. Each of them have their own purpose. There is also grey-box testing which is called when both black and white-box techniques are used.

black-box testing is verifying a system's functionality against its specified requirements without examining the internal structure or implementation details [Mas25]. Testers compare inputs and expected outputs, that simulates user behaviour and ensures that the software behaves as intended. black-box testing also has techniques such as equivalence partitioning, boundary value analysis, decision table testing and others, to make testing more effective[Mas25]. black-box testing is best to use when data handling, interfaces and user interactions is needed to identify flaws in them and make them suitable for requirements-based validation [Mas25].

White-box testing is involving testing the internal logic like source code in software. It requires detailed knowledge of the source code and typically such testing is done by developers. White-box testing is highly effective in detecting logical errors and ensuring that all possible control paths have been exercised, but it demands significant programming knowledge [Mas25].

### 1.1.3. Challenges in manual testing

Manual testing still widely used in industry. It faces several critical challenges that affect efficiency, accuracy, and scalability. One of the significant limitations is the high human effort and cost involved, because manual processes are labor-intensive and time-consuming. It makes manual testing difficult to scale for larger and complex software systems [HEJ+21]. Another challenge is manual test case generation and it remains one of the most resource-demanding tasks. Studies show that around 42% of reviewed research highlights difficulties on creating and maintaining manual test cases, because they lead to incomplete test coverage [Abd20]. Another issue is the integration of manual testing with modern development cycles, such as Agile and DevOps. Shortened release cycles demand faster feedback, but long-running manual test creation can temporary hold development and slow down deployment affecting productivity [HEJ+21]. Organizational and technological barriers also stops the adoption of automation as a solution. Lack of time or budget, often forces teams to rely on manual approaches, even when automation could reduce effort. In some industries, like medical technology, regulations even force to do manual testing and it limits automation potential [HEJ+21].

Manual testing has flaws like high costs, inefficiencies in test case generation, poor scalability with modern development practices and organizational regulations. By addressing these issues we can see that it requires optimization and automation in testing could help solve those issues.

## 1.2. API Testing

### 1.2.1. Nature of APIs

An Application Programming Interface (API) is an interface between software components that enables them to communicate and work together [Gla11]. Rather than exposing the internal workings of a system, an API can define a clear set of rules and operations that other software can use to access its functionality. This makes APIs both a contract and an abstraction, because they specify what can be done, without requiring knowledge of how it is implemented [Gla11].

In current time, the most well known category is the Web API. A Web API exposes resources and services over the internet using standard web protocols, like HTTP. This allows different applications like: mobile apps, websites, and cloud systems—to consume and share data between them. There are several architectural styles of Web API: SOAP, REST, GraphQL, gRPC. The most widely used form of Web API is the REST (Representational State Transfer) architectural style [CCV+21].

Like any other software system, APIs are not immune to faults or unexpected behaviours. To ensure their reliability, APIs require also testing, which is critical part in modern software development. it is therefore important to first understand why API testing plays such a central role.

### 1.2.2. Importance of API Testing

APIs are the backbone of modern software systems. They are like contract, because they define how components and services interact. To ensure their correctness is essential for system to be reliable, integrate seamlessly, and have strong security. Failures at the API level often spread to dependent services, disrupting business-critical processes. This makes it essential for APIs to be tested.

From a reliability perspective, research by Ehsan et al. [EAC+22] points out that the loosely coupled and highly flexible design of RESTful APIs makes them prone to faults, particularly when requests contain invalid inputs or when descriptive documents (like OpenAPI specifications) are incomplete or inconsistent. Without detailed testing, these issues may only be detected at runtime, leading to instability and a potential loss of user trust.

Integration is another major concern. Research by Golmohammadi et al. [GZA23] points out that modern systems are rarely isolated. Instead of that they depend on internal and external APIs to coordinate workflows across organizations. In sectors such as healthcare, banking, and e-commerce, APIs are critical for enabling activities such as patient data exchange, payment processing, and online purchasing. For this reason, regression testing and contract verification are necessary to ensure that changes to an API do not break existing clients.

Security considerations further reinforce the importance of API testing. Research by Ehsan et al. [EAC+22] point out that RESTful APIs are common targets for attacks due to their wide accessibility, with threats ranging from message alteration to improper authentication. They argue that automated security testing frameworks are essential to ensure confidentiality, integrity, and robustness of API based systems.

Also, both researches [EAC+22; GZA23] underline that APIs function as formal contracts: breaking these contracts by altering schemas or responses can severely impact dependent services. In industries with critical requirements, for example financial systems or healthcare infrastructures, such disruptions can result in financial losses, compliance violations, or critical operational downtime.

This is why it is important to test out APIs and it should not be just a technical requirement, it should be a important step to ensure trust, security and sustainable software system.

### 1.2.3. Types of API Testing

API testing can be structured into functional testing and non-functional testing. Also it could be separated by methodological perspective of black-box and white-box approaches. The main difference is that functional and non-functional shows what is being tested, meanwhile black-box and white-box shows how the testing is conducted.

Functional testing ensures that the API behaves according to its specification and business logic. The main types are:

- Unit testing: checks individual endpoints or isolated API components for correctness.
- Integration testing: validates the communication and interoperability between multiple services or APIs.

- System testing: verifies the API's functionality as part of the entire deployed system.
- Acceptance testing: confirms that the API meets stakeholder and business requirements.
- Regression testing: ensures that updates or modifications do not introduce new faults or break existing clients.

Non-functional testing examines quality attributes ignoring the correctness of responses. The main types are:

- Security testing: identifies vulnerabilities such as authentication flaws, data leaks, or susceptibility to denial-of-service attacks.
- Performance and load testing: evaluates responsiveness and scalability under different workloads.
- Robustness testing: assesses how well the API handles malformed, incomplete, or unexpected inputs.

### 1.2.4. REST APIs in Modern Systems

Representational State Transfer (REST) is a software architectural style by Roy Fielding created to guide the design and development of distributed systems, esp. World Wide Web [Fie00]. REST embodies a collection of constraints that govern how applications communicate in network, in a quest to ensure scalability, simplicity, and reliability. REST puts emphasis on a uniform interface in an effort to make communication between client and server straightforward, recommends independent deployment of the components in a bid to enable flexibility, and embrace a layered architecture that accommodates caching, enforces security, and encapsulates legacy systems [Fie00]. Due to these characteristics, REST has been one of the most followed standards in designing APIs and web services, offering a lightweight yet robust platform for interoperability among different systems.

In making it possible to adopt this interoperation, there have been created standards, like the OpenAPI (formerly known as Swagger), among others. The OpenAPI Specification (OAS) provides a consistent, machine-readable way to define REST APIs, significantly increasing documentation, automation, and testing. OAS is also a description format and a contract between API producers and API consumers, detailing endpoints, request parameters, response format, and error codes. This makes it easier for disparate systems and teams to interoperate and helps to manage risks associated with implementation error or misunderstanding. The OpenAPI Specification provides a standardized, language-independent interface to HTTP APIs, thus enabling humans and machines to discover a service's functionality without direct access to source code or by manually-written documentation. When well-defined, an OpenAPI Description (OAD) makes it easier for developers to work with APIs and thus decreases adoption barriers. Indeed, the specification supports a wide variety of use cases, ranging through interactive doc tools automatically generating API references, code generator tools generating client SDKs and server template placeholders in many programming languages, and test frameworks test against the specification. An OAD is made up of an OpenAPI Document, in either JSON or YAML, following the form of the specification. The document is the entry point, and it can point to other files in a bid to define complex systems. The

field paths, which are used in pointing to accessible API endpoints, and the field components, which define values for schemas, security definitions, and response objects are noteworthy features. All features enable repeated, predictable reuse, modularity in design, as well as efficient maintenance of massive APIs. Through the codifying of API design in such a way, OAS facilitates increased compatibility, maintainability, as well as scalability of development processes. This widespread usage within industries shows its pioneer status in facilitating a blend of the architectural principles of REST and the practical aspects of the modern API ecosystem.

## 1.3. REST API Testing Strategies

While testing REST APIs requires tailored methods despite how they are stateless, resource-oriented, and rely on HTTP-based communication, there are two major perspectives in existing literature, namely, the methodological distinction between white-box and black-box testing, and the use of automated methodologies guided by API definitions.

As Golmohammadi et al. [GZA23] describe, black-box testing has been the most prevalent in REST API research, covering approximately 72% of methods. The system under test in this paradigm is tested through its publicly accessible endpoints without direct access to the internal code. Classic methods are fuzzing, in which numerous automatically created requests are submitted in order to test input robustness and handling; property-based testing, in which API responses are checked against formally established properties like idempotency; and specification-based testing, in which contracts like the OpenAPI description are applied to ensure responses conform to announced formats. Tools, like RESTler, Schemathesis, and QuickREST, encapsulate these methods through a combination of specification parsing and automated request creation.

White-box testing, on the other hand, reviews the internal implementation of REST services. Source code instrumentation and code coverage metrics are used by it to direct test generation as well as discover untrodden paths. Tools like EvoMaster incorporate search-based approaches with runtime instrumentation in order to have maximum exercised code and discover faults that are ignored by the black-box approaches. Hybrid methods have also been put forward, by uniting schema-driven black-box testing and white-box heuristics in order to tackle restrictions not formally documented in the specification [GZA23].

In addition to this methodological axis, approaches to testing REST APIs are also categorized along a functional focus. Ehsan et al. [EAC+22] enumerate categories as including test generation based on specification, model-driven testing, property-based testing, metamorphic testing, and regression testing. Specification-based approaches, for example, use OpenAPI or JSON schemas automatically to generate test suites, while model-driven approaches use state machines or Petri nets to check against formal models. Metamorphic testing checks required properties on multiple runs (e.g., successive GET requests repeated are supposed to return the same results), and regression testing checks behaviours against different versions of APIs in order to test backward compatibility.

Among all issues raised in both papers, one of the most significant ones is validating authenticated APIs. Secured end-points, typically authenticated by means of token- or cookie-based mechanisms, make it difficult to implement automated tests since session management and

expiry are unpredictable. The majority of currently in-use strategies either skip or incompletely support such scenarios [EAC+22]. Briefly, REST API testing techniques mainly fall into black-box, white-box, and hybrid techniques, all strongly emphasizing automation and specification techniques. black-box fuzzing and specification tools are overwhelmingly most dominant, but there is a tendency towards combining them with white-box knowledge in an effort to increase coverage, particularly in the case of large or limited APIs.

## 1.4. Comparative Studies on API Testing

Testing a REST API is a main thing of software quality assurance and it is important. It verifies that the service meets the expectations regarding reliability, performance, and security, as well as the specified functionality in various scenarios [BFG+21]. A robust API is often critical to business operations for example in payment, data integration, mobile backends, so undetected issues can have serious consequences. Moreover, web APIs rely on network communication and often interact with databases or external systems, making them complex to validate [BFG+21; GZA23]. Automated testing of REST APIs helps catch defects early to ensures regressions are detected as services evolve, and can validate not only functional behaviour but also non-functional requirements (performance, security, etc.) [BFG+21]. In summary, because REST APIs form the critical link between client applications and server logic, systematic testing is necessary to reveal implementation defects and verify that all requirements (functional and non-functional) are satisfied [BFG+21; GZA23].

### 1.4.1. Functional vs Non-Functional Testing

Testing can be broadly categorized into functional and non-functional testing. Functional testing focuses on whether the API correctly implements its intended operations and business logic. Non-functional testing covers other quality attributes such as performance, scalability, reliability, security, usability. For REST APIs, functional testing often means verifying that each endpoint behaves as specified. It should return the correct status codes and data for valid inputs, and handles invalid or edge-case inputs properly [BFG+21]. Non-functional testing, by contrast, might measure response time, throughput, or stress behaviour of the API under load, or assess security aspects for example authentication, input sanitization [BFG+21]. In practice, many API testing approaches focus on functional requirements firstly, since those are easier to automate and measure. As one recent study notes, that functional testing is the most common type of black-box testing that can be employed for APIs, whereas non-functional aspects (performance, security, etc.) often require specialized tools and are harder to infer from the API specification [BFG+21]. Nonetheless, a comprehensive testing strategy for REST APIs should include both: functional tests validate correct behaviour against the API's specification, while non-functional tests ensure the API meets its quality requirements, for example response time, error handling, security [BFG+21].

### 1.4.2. Black-box Testing of REST APIs

Since the internal code of a deployed REST service is usually not available during testing, most API testing is done in a black-box fashion. In black-box, specification-base, testing, the tester only knows the API's interface: its endpoints, parameters, data formats, and does not inspect the source code. Test cases consist of HTTP requests sent to the API, and testing involves examining the HTTP responses and side effects to check for conformance with the APIs specification [BFG+21]. In other words, each test exercise sends a request (with some method, URL, headers, and input payload) and then verifies that the API returns the expected status code and data according to the spec [BFG+21]. This is why API testing is often called specification-based or black-box testing: the API is treated as a black-box with known inputs and outputs, and the testing validates the output for given inputs without knowledge of internal implementation [BFG+21].

In a black-box context, functional testing focuses on correct requests and responses, while fault-based or negative testing is also common for APIs. Fault-based testing intentionally injects incorrect or out-of-range values to verify that the API handles errors correctly. For example, one approach sends invalid parameter values and checks that the service returns appropriate error status codes, for example from 400 to 499 range, rather than crashing or producing wrong results [BFG+21]. Such negative testing helps ensure robustness, because a test succeeds if the API correctly rejects bad inputs. Recently automated REST-testing tools often include modules for fault-based testing, other could call it "error testing", that changes requests to generate invalid cases [BFG+21].

Overall, automated REST API testing frameworks typically adopt black-box strategies. They take the APIs specification (often an OpenAPI/Swagger file) as input and automatically generate test requests [BFG+21]. This can include generating valid calls for each endpoint, to test functionality, as well as invalid calls, to test error handling. The key is that testing relies only on the interface description, not on source code. This separates black-box API testing from white-box techniques, which require code, such as some model-based or search-based testing tools [BFG+21].

### 1.4.3. Black-box Testing Techniques

A variety of classical black-box testing techniques can be applied to designing REST API tests. These techniques originate from general software testing, but they adapt naturally to the API context. Important examples include Equivalence Partitioning and Boundary Value Analysis, Decision Table Testing, State-Transition Testing, Use-Case/Scenario Testing, Error-Guessing (Fault Injection), and Graph-Based/Model-Based Testing. Each technique provides a different way to obtain test cases from the API specification or requirements. Below are brief overviews of each black-box approach:

- Equivalence Partitioning and Boundary Value Analysis: these are fundamental domain-based techniques. Equivalence partitioning divides an input domain into classes (partitions) of values that are expected to act similarly. For example, if an API parameter "age" accepts from 0 to 120, one might partition this into valid for example. 0–120 and invalid for example below 0 (<0), above 120 (>120) classes. Boundary value analysis focuses on the

edges of these partitions – for example testing age = 0, 1, 119, 120, 121. Together, these techniques aim to reduce the number of test cases while still covering representative values. These classic methods are explicitly named in the standard testing syllabus [HR25], and they remain useful for REST APIs. One can apply them to query parameters, headers, JSON field values, or any other field, to choose proper test inputs. They are especially helpful when parameters have ranges or categories, for example: numeric ranges, enumerated sets. [HR25].

- Decision Table Testing: This technique handles complex business logic that depends on combinations of conditions. A decision table lists all relevant input conditions or parameter value combinations and the expected outcomes or actions. Each column in the table represents one set of conditions (rules), and from that one or more test cases can be derived. In practice, decision tables help testers systematically cover combinations without redundant tests. As one survey notes, decision tables are powerful in finding defects both in implementation and specifications and allow testers to eliminate redundant test cases and reduce the overall test effort [HR25]. For REST APIs, decision tables are useful when a response depends on multiple inputs or flags, then the table helps ensure all logical combinations, especially corner-case combinations, are tested.
- State-Transition Testing: Some APIs maintain state between calls, for example, session-based workflows or transaction flows. State-transition testing models the API as a state machine: nodes represent API states or contexts, and edges represent actions (API calls) that change the state. Test cases are then designed to cover transitions and sequences of calls. This is valuable for stateful services or where the sequence/order of operations that matters. For instance, a ticketing API might require you to authenticate, then create a ticket, then close it and each step depends on the previous state. State-transition techniques ensure that all important state changes (including invalid transitions) are tested. State transition models allow testers to catch and visualize complex system behaviour that changes based on its current state and actions or inputs [HR25], which is especially important for APIs with many possible states.
- Use-Case Testing: This is sometimes also called scenario-based or user-journey testing. A use case describes a sequence of steps or transactions that a typical user or client will perform using the API. Use-case testing constructs test cases to cover these real world scenarios end to end. For example, a use case might be "user signs up, updates profile, then retrieves dashboard data." Each scenario is mapped to a series of API calls with correct data. Academic surveys by Hamburg and Roman [HR25] note that scenario/use-case testing is common in practice for web services [HR25]. Use-case tests ensure that the integrated behaviour of multiple endpoints works correctly together. Coverage requirement for scenarios usually needs at least one test for the main flow, and tests for each alternative flow or exception path [HR25]. Mainly use-case testing validates higher-level, functional workflows of the API.
- Error-Guessing / Fault Injection: In practice, experienced testers often perform error-

guessing, it means intuitively guessing likely mistakes and designing tests accordingly. This is less formal, but it involves thinking of common error cases like missing parameters, SQL injection strings, wrong data types) and testing them. In academic terms, this is similar to fault-based testing or negative testing. For REST APIs, this might mean sending requests with missing or wrong parameters, invalid JSON, excessively large payloads, etc., and checking that the API fails safely (returns the correct error code without crashing). The study emphasizes this approach: fault-based testing... has the purpose of injecting incorrect values in the requests and to validate that the system will respond accordingly without failure [BFG+21]. In other words, these tests are successful if the API correctly handles the errors.

- Graph-Based / Model-Based Testing: This involves building a model, often a graph or UML diagram, of the APIs behaviour and generating tests from it. For example, one could create a cause-effect graph or an activity diagram representing API flows, and then get test cases to cover paths in the graph. Graph-based methods are more commonly discussed for GUI or application testing, but can apply to APIs too, for example testing RESTful choreographies. While less frequently covered in API testing literature, model-based strategies (graphs, state charts) can systematically explore complex interactions. One recent text even mentions graph models in the context of scenario testing, where an activity diagram is used and loop coverage criteria are defined for it[HR25]. In practice, graph-based testing for APIs might mean ensuring that every node (endpoint) and transition (state change) in an API model is exercised by at least one test.

Each of these techniques brings a different perspective. For example, equivalence partitioning and boundary value analysis focus on numeric or text input domains, decision tables focus on logical combinations, state-transition focuses on sequences over time, and use-case focuses on end-to-end behaviours. A comprehensive REST API test suite will often combine several techniques. Choosing the right technique depends on the APIs characteristics and the testers goals: equivalence partitioning might be used for simple parameter validation, decision tables for complex business logic, and scenario testing for critical workflows. The new ISTQB syllabus explicitly groups these techniques as foundational for a Test Analyst role [HR25], reflecting their importance in practice. By applying a mixture of these black-box methods, testers can systematically cover both typical and atypical uses of the API, increasing confidence in its correctness and robustness.

## 1.5. Large Language Models in Test Generation

### 1.5.1. Overview of NLP and LLMs in Software Engineering

Natural Language Processing (NLP) techniques have long been used in software engineering to support requirements analysis and test case generation. Early NLP-based approaches extracted use case elements, actors, and conditions from natural language requirements, often relying on boilerplates, or semi-formal models to mitigate ambiguity [FFV+23; LCS+24]. While these

methods achieved partial automation, they remained limited in handling unstructured or negative requirements.

Large Language Models (LLMs) represent a significant advance by enabling direct mapping from natural language to structured artifacts through transformer-based contextual embeddings [VF25]. Their adoption has extended across requirements classification, requirements tracing, code generation, vulnerability detection, and test case interference [ATT+25; VF25]. Not including requirement analysis, LLMs are increasingly applied in testing themselves, such as in automated evaluation of code completion systems [LWL+23].

### 1.5.2. Capabilities

LLMs have demonstrated the ability to process unstructured requirements and extract meaningful actors, conditionals, and expected behaviours directly from natural language. Unlike previously mentioned NLP approaches, they are less dependent on formalized boilerplates, though templates still enhance reliability [FFV+23; LCS+24]. Persistent challenges remain in interpreting negative requirements, which are particularly error-prone for automated approaches [LCS+24].

LLMs can generate unit tests, acceptance tests, and even complete API call sequences from textual descriptions [ATT+25]. Fischbach et al. tool CiRA for example extracts conditionals from informal requirements and produces acceptance tests automatically [FFV+23]. Recent benchmarks in studies show that fine-tuned LLMs achieve 78.5% syntactic correctness, 67.1% requirement alignment, and 61.7% code coverage when generating tests from requirements [ATT+25].

Both prompt design and fine-tuning critically shape test generation quality. Structured prompts increase correctness and relevance, while fine-tuning on domain-specific requirement–test datasets boosts accuracy and coverage [ATT+25]. In requirements engineering, fine-tuning helps balance general linguistic competence with domain-specific semantics, improving clarity and alignment [VF25].

### 1.5.3. Studies on black-box test generation using LLMs

Several recent studies explore LLMs in black-box contexts.

- Alagarsamy et al. in his article proposed a text-to-testcase benchmark, evaluating LLMs in requirement-to-test generation [ATT+25].
- Previous efforts in tools, such as AthenaTest, A3Test, and ChatUniTest, mainly addressed code-to-test translation, limiting their applicability to test driven development [ATT+25].
- Fischbach et al. proposed tool CiRA applied NLP/LLM techniques for acceptance test case creation [FFV+23], while Lim et al. article focused on boilerplate-driven NLP for extracting tests from positive and negative requirements [LCS+24].
- Li et al. tool CCTEST introduced an automated black-box framework for testing and repairing LLM-based code completion systems [LWL+23]. Using program structure-consistent (PSC) mutations, CCTEST generated prompt variants to detect inconsistent or erroneous completions across tools such as Copilot, GPT-Neo, GPT-J, and CodeGen.

From ∼182k test inputs, it identified 33,540 erroneous outputs and improved completion accuracy by up to 67% through repair strategies [LWL+23].

All these works illustrate the growing ecosystem of LLM-based test generation, spanning requirement-driven test synthesis and meta-testing of LLM outputs themselves.

### 1.5.4. Challenges

Despite these advances, large language models also have several limitations regarding to the reliability of LLM-driven test generation First of them is hallucinations. LLMs may invent APIs, assertions, or conditions not grounded in the requirements [ATT+25; LWL+23]. Another one is lack of determinism. Outputs vary across runs, undermining reproducibility and complicating regression testing [LWL+23; VF25]. Next one is regarding ensuring coverage. Even with fine-tuning, requirement alignment (∼67%) and coverage (∼61%) remain partial [ATT+25]. Another one would be validation of generated tests. Syntactic correctness does not guarantee semantic validity. Testing mutation, human oversight, or frameworks like CCTEST are needed to detect inconsistencies [ATT+25; LWL+23]. Last one would be ambiguity in requirements. Unrestricted natural language, particularly negative requirements, introduces inconsistencies that impede reliable automated test derivation [LCS+24].

## 1.6. Requirements in Software Engineering

### 1.6.1. Functional vs Non-Functional Requirements

Functional requirements (FRs) specifies what a system must do, like its services, behaviours or functions. Non-functional requirements (NFRs) in other hand describe how well the system performs or constraints on those functions. For example, a functional requirement might state "the system shall allow a user to log in," and non-functional requirements specify qualities like performance or security ("login must complete within 2 seconds," or "communication must use encryption"). Recent surveys also mentions this distinction. In his article Cyrille Dongmo notes that "functional requirements describe the services to be rendered by the software," while NFRs define the quality of service and constraints on those functions [Don24]. Similarly, in article by Tahir et al. author defines FRs as system behaviours or actions (the "what") and NFRs as qualities such as performance, availability, security, usability, etc. (the "how well") [TJT+25]. In reality, the two are complementary: any FR feature is constrained or qualified by NFRs, like safety or reliability. The literature typically lists examples of FRs, as functions or user actions, versus NFRs, like quality attributes, for example usability or scalability, to illustrate the difference [Don24; TJT+25].

### 1.6.2. Natural Language in Requirements Specification

Natural language (NL) specification is by far the dominant approach in industry, because it is immediately accessible to all stakeholders. In an extensive industry study, Franch et al. [FPQ+23] observed that "natural language constitutes the main specification artefact" and is "the easiest

option for practitioners to choose" when writing requirements. Article by Antoniou et al. [AB23] similarly note that a major advantage of NL is that "stakeholders do not need to learn anything new to use this technique" [AB23] making requirements documents immediately understandable to customers, managers, developers, and testers.

Sadly, NL suffers from well known weaknesses. Nearly all recent sources emphasize its ambiguity and lack of accuracy. In article by Franch et al. [FPQ+23] reports that NL requirements suffer from a range of problems, including ambiguity, inconsistency, and incompleteness. Article by Necula et al. [NDG24] reinforce this, noting that natural language is prone to ambiguity, leading to incompleteness and inconsistencies in requirement specifications. At the same article by Necula et al. [NDG24] explicitly warn that the danger of NL is ambiguity. Ambiguous wording and multiple valid interpretations are uncontrolled in pain text requirements. These verbal issues make NL easy for stakeholders to write but hard for tools or engineers to interpret without error. From this we can see, that NLs strength is its accessibility to people, where nothing should be learned, but its weakness is imprecision: studies consistently report that ambiguity and inconsistency in NL requirements are the top practical challenges in specification[FPQ+23; NDG24].

### 1.6.3. Standards, Frameworks, and Templates for Requirements

To mitigate NL problems, researchers and practitioners have proposed structured templates and notations. A requirement template, also called a boilerplate, provides a fixed syntactic pattern with placeholders (for example: "The <system> shall <behaviour> <object> [when <condition>]"), which a requirements engineer fills in. Pohl and Rupp in 2011 define a template as "a blueprint for the syntactic structure of individual requirements," essentially the same as a boilerplate [AB23]. Using templates helps reduce ambiguity and enforce consistency. A classic example is the Rupp template (introduced by Pohl and Rupp), which structures a requirement into categories like obligation, action, subject, object, and conditions. Article by Antoniou et al. [AB23] describes Rupps method and show how templates specify requirement main idea and its conditions in a gradual process. Article by Hull et al. (2010) called these templates "boilerplates" – fixed patterns that the engineer fills in to standardize NL [AB23].

Another modern template is EARS (Easy Approach to Requirements Syntax). EARS defines five simple sentence patterns for functional requirements:

- Ubiquitous ("The <system> shall ..." – always true);
- Event driven ("When <event>, the <system> shall ...");
- Unwanted behaviour ("If <trigger>, then the <system> shall ...");
- State driven ("While <state>, the <system> shall ...");
- Optional feature ("Where <condition> holds, the <system> shall ...").

These are all boilerplates that capture common functional forms. Article by Antoniou et al. explains that EARS template distinguishes 5 different types of requirements (ubiquitous, event driven, unwanted, state, optional), and Arora et al. describe using EARS to classify and normalize requirements[AB23]. By forcing usage of requirements into these patterns, then EARS reduces wording complexity and guides authors to be explicit.

Industry also uses more general frameworks, like for example, the Volere Requirements Template. It is a popular SRS template by Atlantic Systems Guild that specifies sections for business rules, scenarios, and details. International standards set out recommended content and structure for SRSs, though surveys find that most companies do not strictly follow a fixed standard[FPQ+23]. Article by Franch et al. reports that while most requirements specifications use templates or guidelines, these are hardly ever formal standards[FPQ+23], because practitioners often adapt ad hoc templates. Regardless, using any semi formal template (Volere, EARS, Rupp, UML based systems models, controlled languages, etc.) is shown to improve clarity. The literature lists EARS and Rupp as leading structured template methods. These approaches all aim to constrain NL with fixed syntax , as this makes the resulting requirements easier to analyze or even automatically check [AB23].

### 1.6.4. From Natural Language Requirements to Test Cases Challenges

Translating NL requirements into executable test cases is difficult. The literature identifies multiple obstacles. Firstly, NL requirements are often incomplete or underspecified, so a tool trying to generate test cases has to fill gaps. As Ahmad et al. note in article [AIA+19], most requirements are expressed in NL, which can face lexical ambiguities, contextual inconsistencies, and structural incompleteness [YHC+25]. When requirements are ambiguous or incomplete, any derived test cases could be unreliable or miss critical behaviour. Efforts to formalize NL requirements, for example by mapping to logic or UML, require substantial manual effort and can lose information. Article by Ahmad et al. [AIA+19] reports that translations to semi formal models often require substantial manual effort, risk losing important information (such as non-functional requirements), and do not guarantee that the requirements will be complete and unambiguous[YHC+25]. In practice, domain knowledge and human intervention remain essential to resolve vagueness before testing. Secondly, even once requirements are formalized, generating concrete (executable) test cases is hard. Most automated approaches stop at abstract test descriptions. Converting those into full test scripts requires mapping abstract steps to code calls or input values, which is often done semi manually. In article by Ahmad et al. [AIA+19] summarizes this as a gap between“abstract test cases (derived patterns or flows) and truly executable test scripts[YHC+25]. In short, current requirements based test generation methods primarily produce abstract test cases that lack sufficient detail to run without further elaboration[YHC+25].

Sadly, but tool and process gaps persist. Few off-the-shelf tools can seamlessly parse NL requirements and generate tests. Most research prototypes cover limited patterns or domains. As the Franch et al. industry study notes, requirements, artifacts used in this activity specification, are usually just text, and few firms leverage automated test generation tools [FPQ+23]. Overall, the agreement is that linguistic challenges: ambiguity, polysemy, incomplete context, and technical challenges, like lack of standardized NL formats and NLP capabilities, make end-to-end automatic test generation an open problem [YHC+25]. Researchers emphasize that high quality, precise requirements, like expressed in controlled language or templates are a prerequisite for generating

reliable tests [AB23; YHC+25]. Until NL issues are resolved, converting requirements to formal test cases will continue to require significant manual effort and iterative review.

## 1.7. Summary and Research Gaps

The literature highlights that software testing remains essential for ensuring reliability, security, and compliance, with black-box methods being dominant in REST API testing. These methods—ranging from specification based to fuzzing and property based testing rely heavily on formal interface descriptions such as OpenAPI. Different tools automate request generation and validation, but challenges remain with authenticated endpoints, stateful interactions, and non-functional aspects.

On the requirements side, natural language specifications continue to be the primary medium in industry due to accessibility. However, ambiguity and incompleteness frequently hinder their direct use in automation. Structured templates, like EARS and Rupp, improve clarity but are inconsistently applied in practice.

Recent advances in natural language processing and large language models (LLMs) have shown promise in bridging requirements and testing. LLMs can parse unstructured requirements and generate black-box test cases that resemble realistic API calls. Yet, generated outputs often remain abstract, lack determinism, and only partially align with intended requirements. As such, the transition from natural language requirements to fully executable REST API test suites remains unresolved.

### 1.7.1. Identified Research Gaps

From the reviewed studies, several gaps emerge in the context of requirements driven REST API testing:

1. End-to-end automation: Current approaches do not achieve a complete pipeline from natural language requirements to structured representation then executable black-box REST API tests.
2. Executable test generation: Most requirements based methods stop at abstract scenarios, requiring manual effort to map to concrete API requests and assertions.
3. Ambiguity in requirements: Natural language introduces inconsistencies that limit reliable automation, while structured templates are underutilized in real-world projects.
4. Limitations of LLMs: Although LLMs improve requirement to test mapping, they remain prone to hallucination, nondeterminism, and coverage gaps, especially for negative and stateful scenarios.
5. Integration into development workflows: Existing tools focus on code or specification levels and rarely integrate requirement based black-box testing into Agile/DevOps pipelines.

### 1.7.2. Justification for Proposed Contribution

To address these gaps, this thesis focuses on automating the creation of black-box REST API tests directly from requirements. By leveraging structured requirement representations and LLM enhanced techniques, the goal is to move beyond abstract test case generation toward concrete, executable API level tests. Such an approach would reduce manual effort, enhance coverage, including error handling and stateful interactions, and enable more seamless adoption of requirements driven testing in modern development environments.

# 2. Methodology

## 2.1. Research Approach

This work approach is to design and build suggested method and then test it out with experiment and evaluate. A software prototype was designed and implemented to automatically generate executable system level tests from natural language requirements and an OpenAPI specification. The prototype serves as the main research tool. Effectiveness is evaluated experimentally by generating and executing tests for a fixed dataset of requirements and analyzing the obtained results.

## 2.2. Process Design

The proposed approach follows a structured processing pipeline that transforms natural language functional requirements into executable system level tests for REST APIs. The pipeline is designed to operate in a fully automated approach, without manual intervention during test generation or execution.

Since the system under test is a REST API, a testing framework capable of constructing HTTP requests and validating responses is required. Pytest was selected as the test execution framework because it integrates naturally with the Python based implementation of the proposed approach and provides expressive assertion mechanisms suitable for API testing.

The overall process begins with the selection of a target API and its associated requirement dataset. Then followed by configuration of the execution environment, including authentication credentials if required and base URLs. The OpenAPI specification is then loaded to provide structured contextual information for test generation.

Test generation is performed step by step for each requirement. For every requirement, a generation context is constructed that combines the requirement text, the OpenAPI specification, and additional execution context. A large language model is used to generate candidate pytest test code, which is subsequently validated to ensure syntactic correctness and compliance with predefined structural rules. If validation fails, the generation step is retried up to a predefined limit.

Once valid test code is obtained, the generated tests are saved and executed automatically using pytest. During execution, HTTP requests are sent to the target API and responses are collected. Execution outcomes, including pass or fail results, are recorded for later analysis.

Finally, execution results are aggregated across all requirements to compute evaluation metrics and support subsequent qualitative and quantitative analysis. The complete workflow of the proposed approach is illustrated in Figure 1.

The overall process consists of the following steps:

1. Selection of the target API and its associated natural language requirement dataset.
2. Configuration of the execution environment, including authentication credentials and base URL.

3. Loading of the OpenAPI specification and initialization of the test generation prototype.
4. Iterative processing of requirements, where for each requirement:
   (a) A generation context is constructed from the requirement text and OpenAPI specification.
   (b) Candidate pytest test code is generated using a large language model.
   (c) Generated code is validated against syntactic and structural rules, with retries applied if validation fails.
   (d) Valid test code is saved and executed automatically using pytest.
   (e) Execution results are collected and recorded.
5. Aggregation of execution results and computation of evaluation metrics.

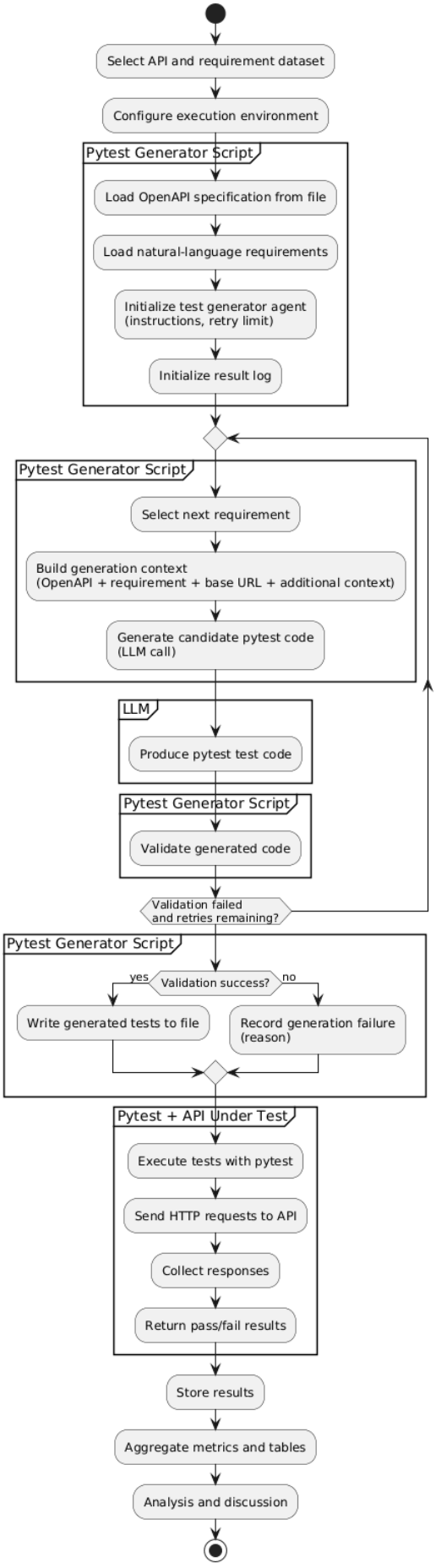


Figure 1 Activity diagram for process approach

## 2.3. Data Sources

For testing purposes, system requirements and OpenAPI specification are needed. This requires to select some working apis. During testing, it was observed, that demo apis are not good choice for this work, because usually they have no logic underneath and in some cases, like "Petstore API", they do not follow defined OpenAPI specification. For example defined "Pet" data object, required to call POST request for creation of a new pet in the store "/pet", requires mandatory field "name", but making request without it still passes through and returns success status code. This is why it was decided to use real apis. So some available apis were selected like FixerAPI and PayPal Order API.

### 2.3.1. Requirements Dataset

The evaluation uses a dataset of functional requirements written in natural language. The requirements were created from the OpenAPI specification of the target API and describe expected system behaviour, error handling and other system scenarios. Below there is how one of requirements looks like:

> *When a client provides a valid access key, the system shall return the list of supported currency symbols.*

The requirements are written in free form natural language rather than using a controlled or formalized syntax. This design choice reflects common industry documentation practices and allows evaluation of the approach under real conditions. Although the requirements were formulated by the author, they were derived from publicly available API documentation, OpenAPI specifications, and observed API behaviour. To improve realism, requirements were additionally reviewed and manually validated against the original API documentation and expected API responses.

To support reproducibility of the experiments, the requirement datasets, OpenAPI specifications, and execution script used were organized together with the prototype implementation in a public source code repository[1]. The repository contains the datasets used during evaluation, together with example execution configurations and instructions for reproducing the experimental workflow.

### 2.3.2. OpenAPI Specification

The OpenAPI specification provides a structured description of the system under test. It defines available endpoints, supported HTTP methods, request parameters, authentication mechanisms, and expected responses.

In the proposed approach, the OpenAPI specification is used to narrow test generation, reduce ambiguity during requirement interpretation and support validation of generated test cases. If system behaves not as defined OpenAPI specification, then either OpenAPI specification is wrongly defined or system works faulty, like in "Petstore API" example.

[1] `https://github.com/Yar3k/system-test-gen`.

## 2.4. System Design

The prototype is implemented as a single Python script that orchestrates test generation using an agent based interface and validates the generated output before saving it as executable pytest tests. Figure 2 presents the component view of the implementation and shows the main modules, data objects, and external dependencies used during test generation.

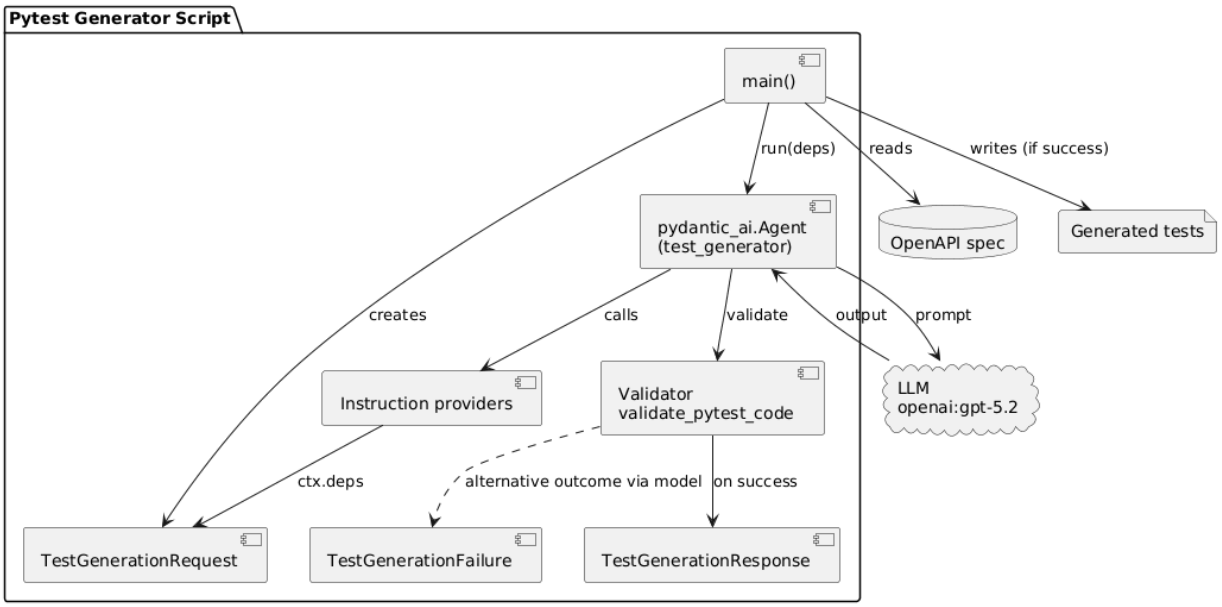


Figure 2 UML Component diagram of implemented prototype

The main elements of the system are:

- Main orchestration (***main()***)
- Input and output data models
- Test generation agent (***pydantic_ai.Agent***)
- Instruction providers
- Output validator (***validate_pytest_code***)

### 2.4.1. Main orchestrator

The main function controls the overall execution flow. It loads the OpenAPI specification from a local file, constructs a TestGenerationRequest object, invokes the test generation agent, and writes the generated pytest tests to an output file when generation succeeds.

### 2.4.2. Input and output data models

The prototype uses typed models to represent inputs and outputs:

- ***TestGenerationRequest*** contains the requirement, OpenAPI specification, API base URL, and optional additional context.
- ***TestGenerationResponse*** contains the generated pytest code.
- ***TestGenerationFailure*** represents unrecoverable generation failure and includes an explanation.

### 2.4.3. Test generation agent

The ***test_generator*** agent is responsible for producing test code from the provided inputs. It receives the request object as dependencies and uses fixed instructions to guide generation of complete, executable pytest tests. As LLM prompt, it uses latest OpenAI GPT-5.2 model

### 2.4.4. Instruction providers

Instruction providers expose request specific information (requirements, OpenAPI specification, API URL, additional context) to the agent via the execution context from helper instructions. This design keeps prompt construction consistent and makes the inputs explicit and reproducible.

### 2.4.5. Output validator

To improve reliability, the generated test code is validated automatically before it is accepted. In validation step, first what is checked is if the output contains no markdown formatting. In some cases LLMs add this markdown formatting to show that this is code part. After that it is checked, if at least one pytest test function is present. After that it checks if required imports are included. In our case 2 imports are required: ***pytest*** for test creation and execution as frameword and ***requests*** for building HTTP requests and executing them. Final validation is validation, if code is valid Python code. Since we do not want to run it, to check if its valid, because it will make requests, so the code is checked syntactically, if it is valid Python.

If validation fails, test generation is retried automatically. If repeated attempts fail, the generation process is terminated and recorded as a failure.

### 2.4.6. Test Execution

Validated test cases are saved as python files. They can be executed using pytest library. Execution results, including pass/fail status and runtime errors, are recorded for later analysis.

No manual modifications are needed for generated tests to be able for execution.

## 2.5. Prototype Implementation

The proposed approach was implemented as a Python based prototype[2]. The prototype uses typed data models to represent generation inputs and outputs together with an agent based abstraction for interaction with the large language model. Three primary models were defined for the implementation: `TestGenerationRequest`, `TestGenerationResponse`, and `TestGenerationFailure`. Since the generated output consists of executable pytest test code, automatic validation was introduced to improve reliability of generated responses. The validation process consists of five main checks:

- Verification that the generated output does not contain markdown code fences (```).

[2] `https://github.com/Yar3k/system-test-gen`.

- Verification that at least one pytest test function is present by checking for the `def test_` pattern.
- Verification that the pytest library import is included.
- Verification that the requests library import is included.
- Verification that the generated code is syntactically valid Python code using Abstract Syntax Tree (AST) parsing through Python's `ast` library.

If validation fails at any stage, the generation process is retried automatically up to five times. If all retry attempts fail, a `TestGenerationFailure` response is returned. Then AI agent is needed. For this prototype, PydanticAI library was used, to be able generate structured responses based on defined model. OpenAI GPT-5.2 was also selected as LLM model, since it is most recent from OpenAI at the moment. The agent was configured with a predefined system instruction describing the expected structure and behaviour of generated tests. The primary instruction used during experimentation is shown below:

```
"You are a pytest test generation expert. "
"Given natural language requirements, generate complete, ready to run pytest
test code. "
"Generate actual test implementations, not just placeholders. "
"Include: imports, fixtures if needed, test functions with real assertions. "
"Return valid Python code that can run directly with pytest. "
"Make tests realistic and cover the main scenarios of the requirement."
```

To expose request data to the agent, additional instruction provider functions were implemented using PydanticAI instruction decorators. These functions provided access to the OpenAPI specification, requirement text, API URL, and optional additional execution context such as authentication information. Generated pytest tests were saved as Python files and later executed using pytest.

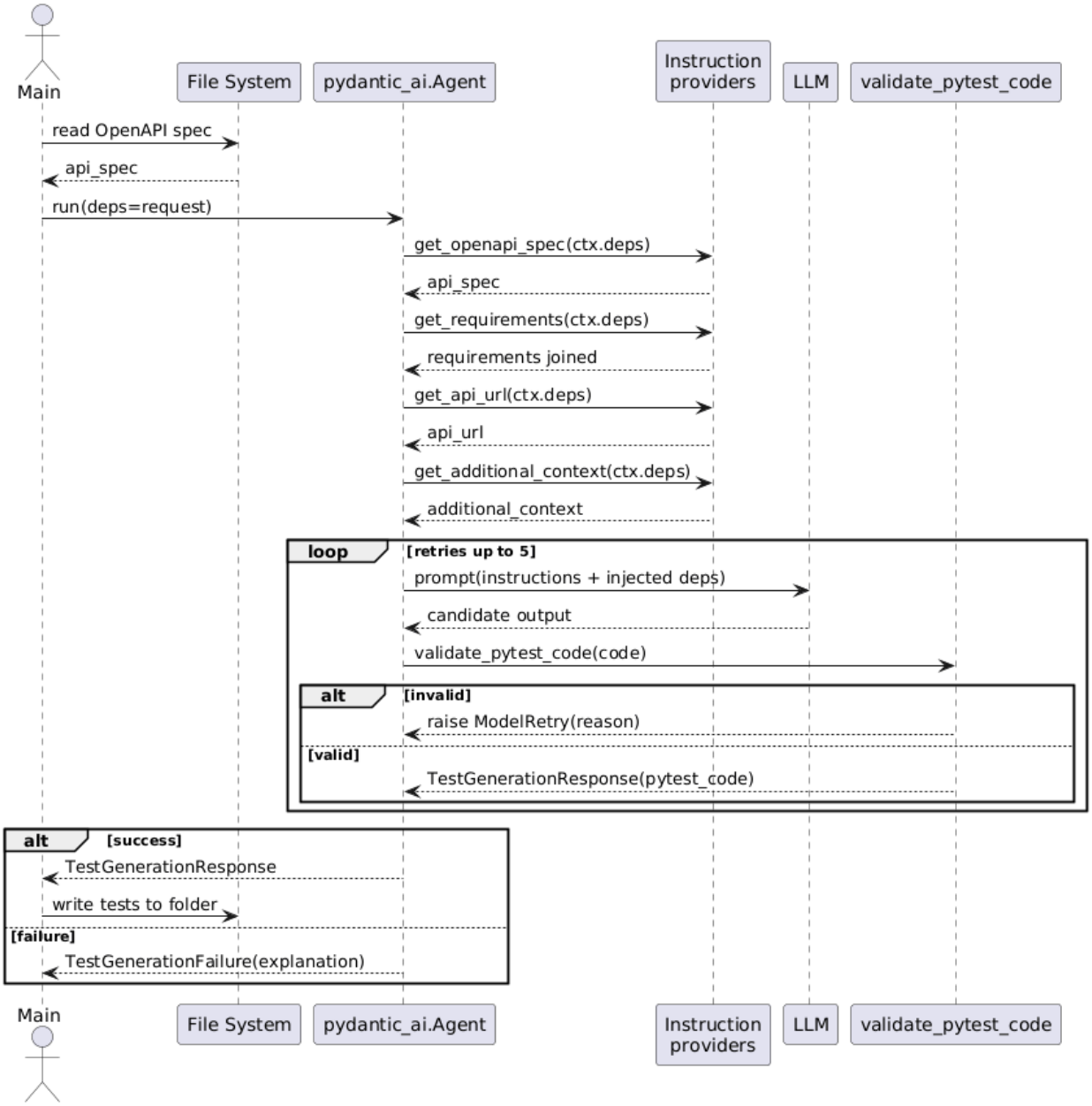


Figure 3 UML Sequence diagram of implemented prototype

Figure 3 illustrates the execution flow of the implemented prototype. The process begins with loading the OpenAPI specification and constructing the request model. Using instruction provider functions, the agent retrieves the OpenAPI specification, requirement text, API URL, and additional execution context if required, for example authentication information. The agent then generates pytest test code, which is validated using the previously described validation process. If validation fails, generation is retried up to the predefined retry limit. Otherwise, the validated pytest code is returned and saved as an executable test file.

The prototype implementation was developed in Python 3.14 using the pytest testing framework and the PydanticAI library for structured interaction with large language models. Test generation experiments were executed on a local development environment running Windows 11. The prototype implementation and experimental artifacts used in this work are publicly available in the accompanying source code repository.

The OpenAI GPT-5.2 model was used for test generation through the OpenAI API. Generation requests were executed using a deterministic configuration with low temperature settings in order to reduce output variability and improve reproducibility of generated tests. The generation process included automatic retry handling, where invalid outputs failing structural or syntactic validation were regenerated until the predefined retry limit was reached.

Generated tests were validated using Python syntax parsing and structural checks before execution. Test execution was performed using pytest with isolated requirement level test files. All experiments were executed using the same OpenAPI specifications, API credentials, and runtime configuration throughout the evaluation process.

## 2.6. Test Generation Strategy

The test generation strategy focuses on producing system level tests that directly reflect the intent of the input requirements.

For each requirement, the prototype attempts to generate one or more test cases that:

- Target the relevant API endpoint.
- Use appropriate HTTP methods and parameters
- Include assertions based on expected responses.

Where possible, negative scenarios such as invalid input or missing authorization are also generated, depending on the content of the requirement.

# 3. Experiment and Results

## 3.1. Experimental Setup

The experiment was conducted using a REST API production system defined by its OpenAPI specification. A dataset of natural language requirements was used as input for test generation. Real world production REST APIs rarely provide publicly available natural language requirement specifications suitable for research purposes. In most cases, only technical API documentation and OpenAPI specifications are available. Because of this limitation, the requirement dataset used in this research was manually constructed based on the publicly available OpenAPI specifications and official API documentation of the evaluated systems.

Large language models were used only as an assisting tool during requirement drafting and reformulation, while the final requirements were manually reviewed and adjusted by the author to ensure consistency with the documented API behaviour. This approach was necessary in order to simulate realistic requirement driven testing scenarios while maintaining alignment with the actual functionality of the evaluated APIs. The requirements describe expected system behaviour, error handling, authorization scenarios, and state dependent operations.

The proposed prototype was executed in a controlled environment using Python. Generated tests were produced in the form of pytest test cases and executed without manual modification. All experiments were performed using the same configuration to ensure consistent results.

## 3.2. Experimental Procedure

The experimental evaluation followed a fixed and repeatable procedure:

1. A subset of requirements was selected from the prepared dataset.
2. The OpenAPI specification of the target API was loaded into the prototype.
3. Executable pytest test cases were generated automatically for each requirement.
4. Generated test cases were validated using automated checks.
5. Validated test cases were executed against the target API.
6. Execution results and errors were recorded for analysis.

No manual changes were applied to the generated tests during the experiment. This ensured that the results reflect the actual capabilities and limitations of the proposed approach.

## 3.3. Evaluation Metrics

The effectiveness of the proposed approach was evaluated using a combination of qualitative and quantitative metrics suitable for black-box testing of REST APIs. Classical white-box testing metrics, such as code coverage or mutation score, are not applicable in this context due to the lack of access to internal system implementation. The selected metrics therefore focus on requirement representation, API surface coverage, executability, semantic alignment, and execution stability of generated tests.

- **Requirement test traceability and coverage**: assessment of whether each natural language requirement is deterministically mapped to one or more generated test cases, preserving a direct traceability relationship between requirements and test artifacts.
- **Endpoint coverage**: assessment of whether the generated test cases exercise the API endpoints defined in the OpenAPI specification, indicating coverage of the exposed API surface.
- **Test executability rate**: evaluation of whether generated test cases can be executed without manual modification, including syntactic correctness and compatibility with the selected test execution framework.
- **Failure diagnostic value**: qualitative assessment of whether test failures provide meaningful information about system behaviour, such as specification mismatches, authorization constraints, or state dependent limitations.
- **Execution stability**: assessment of the consistency of test execution outcomes across repeated runs, with particular attention to variability caused by external API constraints.
- **Efficiency**: evaluation of test generation and execution time, measured as total execution duration and average time per requirement.
- **Semantic correctness**: qualitative assessment of how accurately generated executable tests reflect the behavioural intent of the original natural language requirements, including endpoint selection, request structure, assertions, and expected response validation.

These metrics were selected to evaluate both the technical correctness and the practical usefulness of the generated tests, while also assessing the degree to which generated executable tests preserve the semantic intent of the original natural language requirements.

## 3.4. Test Generation Results

### 3.4.1. Fixer API

The proposed approach was applied to a dataset of requirements derived from the Fixer API OpenAPI specification. For this api 37 requirements were created.

| Metric | Value |
|---|---|
| Total requirements | 37 |
| Functional requirements | 15 |
| Authorization requirements | 9 |
| Error handling requirements | 11 |
| Boundary requirements | 2 |

Table 1. Fixer API requirement dataset overview

From Table 1 we can see what types of requirements this dataset contains. For most requirements, multiple pytest test cases were generated, including positive and negative scenarios. In several cases, more than three test cases were produced from a single requirement, particularly

for error handling and authorization related requirements. From Table 2 we can see that out of 37 requirements, 189 test cases were generated. Out of them 146 passed and 43 failed. Average test per requirement is 5,1

| **Metric** | **Value** |
|---|---|
| Number of requirements | 37 |
| Generated test cases | 189 |
| Average tests per requirement | 5.1 |
| Passed test cases | 146 |
| Failed test cases | 43 |

Table 2. Test generation and execution results for the Fixer API

Requirements that depended on unavailable execution conditions, such as missing authentication credentials or unsupported subscription features, were marked as skipped during execution. This ensured that test execution failures were not misclassified as generation errors.

Overall, the test generation process successfully produced executable pytest test files for all selected requirements.

### 3.4.2. PayPal Checkout Orders API

The proposed approach was additionally evaluated using the PayPal Checkout Orders API. For this api, 30 requirements were created. We can also see what type and quantity of requirement we have from Table 3.

| **Metric** | **Value** |
|---|---|
| Total requirements | 30 |
| Functional requirements | 11 |
| Authorization requirements | 8 |
| Error handling requirements | 9 |
| State dependent requirements | 2 |

Table 3. PayPal Checkout Orders API requirement dataset overview

A dataset of requirements covering order creation, retrieval, update, authorization, capture, and tracking operations was used as input for test generation.

For each requirement, the prototype successfully generated one or more executable pytest test cases. In many cases, multiple tests were produced to cover different authorization states, request payload variants, and error scenarios. Requirements involving complex state transitions or restricted permissions resulted in a higher number of generated tests. As we can see in Table 3, 102 test cases were generated, from them 91 passed and 11 failed. It seems like a better pass/fail ration compared with Fixer API result ratio.

| Metric | Value |
|---|---|
| Number of requirements | 30 |
| Generated test cases | 102 |
| Average tests per requirement | 3.4 |
| Passed test cases | 91 |
| Failed test cases | 11 |

Table 4. Test generation and execution results for the PayPal Checkout Orders API

Overall, test generation was successful for all selected PayPal requirements, demonstrating that the approach works on other apis, even with different authentication.

## 3.5. Test Executability

### 3.5.1. Automated Test Executability

All generated test cases were executed using the `pytest` framework without manual modification. Automated test executability was evaluated based on whether generated tests could be executed directly and produced a clear pass, fail, or skip outcome without requiring human intervention.

For both evaluated APIs, all generated test files were syntactically valid and executable. No manual corrections were required prior to execution, indicating that the test generation and validation pipeline successfully ensured compatibility with the selected test execution framework.

Execution outcomes included both successful and failed test cases. For the Fixer API, a substantial portion of tests executed successfully, while several tests failed due to discrepancies between expected behaviour derived from the OpenAPI specification and the API's actual runtime behaviour. Common execution outcomes included:

- successful execution with correct assertions,
- assertion failures caused by unexpected HTTP status codes,
- failures related to subscription plan restrictions,
- failures caused by inconsistent or proprietary error code usage.

For example, tests generated to validate malformed parameter handling expected HTTP 400 or 422 responses, whereas the API frequently returned proprietary error codes such as 105, 201, or 302. These cases resulted in assertion failures despite the API correctly rejecting invalid input.

A similar pattern was observed for the PayPal Checkout Orders API. All generated PayPal related test cases were executed using `pytest` without manual modification, resulting in a combination of passed, failed, and skipped tests. A large proportion of tests executed successfully, particularly those targeting:

- basic order creation and retrieval,
- schema validation errors,
- handling of non existing resources,

- detection of authorization failures.

Execution failures for the PayPal API were primarily caused by:

- unexpected response payload structures,
- state dependent behaviour not fully captured in individual requirements,
- restrictions related to live payment processing and sandbox limitations.

For instance, tests expecting a specific `intent` value in the order creation response failed when the API returned a response structure different from that anticipated by the OpenAPI specification. Similarly, authorization related tests failed due to varying error response formats across different authentication failure scenarios.

### 3.5.2. Execution Stability (Flakiness)

Execution stability, commonly referred to as test flakiness, describes whether test cases exhibit inconsistent outcomes across repeated executions without changes to test logic. To evaluate execution stability, the generated test suites were executed repeatedly under unchanged conditions.

All generated test cases were executed five times for both evaluated APIs. Across these repeated runs, test outcomes remained consistent: tests that passed or failed did so deterministically in every execution. No instances were observed where a test alternated between passing and failing across runs.

The only prerequisite for repeated execution was the renewal of subscriptions or authorization credentials where required, in order to prevent interference from previously executed tests or expired authentication state. This step ensured that test isolation was maintained and that execution conditions remained comparable across runs.

These observations indicate that the generated tests exhibit full execution stability. No flakiness attributable to non deterministic test logic, request construction, or assertion behaviour was identified. All observed failures were reproducible and consistently linked to external API constraints or specification mismatches.

## 3.6. Endpoint coverage

Endpoint coverage evaluates whether the generated test cases exercise the API endpoints defined in the OpenAPI specification. As discussed in the evaluation metrics section, two interpretations of endpoint coverage are considered:

- Endpoint reachability coverage, which considers an endpoint covered if at least one generated test invoking that endpoint executed successfully.
- Complete endpoint test coverage, which considers an endpoint fully covered only if all generated tests targeting that endpoint executed successfully.

This allows separation between the capability of the proposed approach to reach all API endpoints and the impact of execution constraints on test outcomes.

### 3.6.1. Fixer API

The Fixer API specification defines six distinct endpoints. As shown in Table 5, generated test cases invoked all six endpoints, and each endpoint was exercised by at least one passing test. Under the endpoint reachability definition, coverage therefore reached 100%.

When applying the stricter definition of complete endpoint test coverage, only the `GET /symbols` endpoint was fully covered, as all thirteen generated tests targeting this endpoint passed successfully. The remaining endpoints were classified as partially covered due to one or more failed tests.

Failures for endpoints such as `GET /latest`, `GET /{date}`, and `GET /convert` were primarily caused by undocumented error semantics, subscription plan limitations, and discrepancies between expected and actual response codes. These failures affected complete coverage but did not prevent successful invocation of the endpoints.

Overall, out of six endpoints, one endpoint was fully covered and five were partially covered under the complete coverage definition.

| **Endpoint** | **Tests** | **Passed** | **Failed** |
|---|---|---|---|
| GET /symbols | 13 | 13 | 0 |
| GET /latest | 35 | 26 | 9 |
| GET /{date} | 16 | 13 | 3 |
| GET /convert | 25 | 13 | 12 |
| GET /timeseries | 44 | 39 | 5 |
| GET /fluctuation | 56 | 42 | 14 |
| **Total** | 189 | 146 | 43 |

Table 5. Fixer API complete endpoint test coverage

### 3.6.2. PayPal Checkout Orders API

The PayPal Checkout Orders API defines nine endpoints related to order creation, retrieval, update, authorization, capture, and tracking. As shown in Table 6, generated tests successfully invoked all nine endpoints at least once. Consequently, endpoint reachability coverage for the PayPal API also reached 100%.

Under the complete endpoint test coverage definition, six endpoints were fully covered, including `PATCH /orders/{id}`, `POST /orders/{id}/capture`, and all tracking related endpoints. Three endpoints were classified as partially covered due to failed tests, namely order creation, order retrieval, and order authorization.

The observed failures were largely attributable to authorization constraints, sandbox limitations, and implicit workflow dependencies, rather than incorrect test generation. As a result, while full execution success was not achieved for all endpoints, the approach demonstrated the ability to generate and execute tests across the entire API surface.

| Endpoint | Tests | Passed | Failed |
|---|---|---|---|
| POST /v2/checkout/orders | 14 | 9 | 5 |
| GET /v2/checkout/orders/{id} | 13 | 8 | 5 |
| PATCH /v2/checkout/orders/{id} | 13 | 13 | 0 |
| POST /v2/checkout/orders/{id}/authorize | 9 | 8 | 1 |
| POST /v2/checkout/orders/{id}/capture | 14 | 14 | 0 |
| POST /v2/checkout/orders/{id}/track | 14 | 14 | 0 |
| PATCH /v2/checkout/orders/{id}/track | 10 | 10 | 0 |
| GET /v2/checkout/orders/{id}/track | 8 | 8 | 0 |
| DELETE /v2/checkout/orders/{id}/track/{tid} | 7 | 7 | 0 |
| **Total** | 102 | 91 | 11 |

Table 6. PayPal Checkout Orders API complete endpoint test coverage

#### 3.6.3. Discussion

The endpoint coverage results demonstrate that the proposed approach is capable of exercising the full set of endpoints defined in the OpenAPI specifications for both evaluated APIs. Achieving 100% endpoint reachability coverage for both APIs indicates that the combination of natural language requirements and OpenAPI context is sufficient to generate tests that interact with all exposed endpoints.

At the same time, the difference between reachability coverage and complete endpoint test coverage highlights the influence of external execution constraints, such as subscription plans, authorization requirements, and state dependent workflows. These constraints affect execution outcomes but do not limit the ability of the approach to generate meaningful endpoint level tests.

### 3.7. Requirement Coverage and Alignment

#### 3.7.1. Requirement Coverage

##### 3.7.1.1. Fixer API

Requirement coverage was defined as the ability to generate and execute at least one test case for a given requirement. Using this metric, the proposed approach achieved high coverage across functional, authorization, and error handling requirements.

Some authorization related requirements were intentionally skipped during execution due to missing or invalid access credentials, while still being considered covered at the test generation level.

##### 3.7.1.2. PayPal Checkout Orders API

Requirement coverage for the PayPal API was high, as executable tests were generated for all functional, authorization, error handling, and state based requirements. Some requirements

were marked as skipped during execution due to external constraints, such as operations requiring live payment approval or elevated permissions.

### 3.7.2. Requirement Test Alignment

#### 3.7.2.1. Fixer API

Requirement–test alignment was assessed by examining whether the generated tests correctly reflected the intent of the corresponding requirement, independent of execution outcome.

The majority of generated tests were found to be aligned with their requirements. Execution failures were most often caused by: undocumented API behaviour, subscription plan constraints not fully described in the OpenAPI specification, ambiguous requirements that allowed multiple interpretations.

For instance, several tests expected HTTP 403 responses for forbidden operations, while the API returned HTTP 400 responses with custom error codes. In these cases, the generated test logic reflected the requirement intent, but the assertion did not match the API's actual behaviour.

#### 3.7.2.2. PayPal Checkout Orders API

Requirement–test alignment was evaluated independently of execution success. Most generated tests accurately reflected the intent of their corresponding requirements.

Misalignment was primarily observed in state dependent requirements. For instance, authorization tests failed when the order was in a "PAYER_ACTION_REQUIRED" state rather than the expected unapproved state. In such cases, the generated test logic correctly followed the requirement description, but runtime behaviour depended on implicit workflow constraints not explicitly stated in the requirement.

These results highlight the challenge of interpreting stateful requirements without additional workflow context.

## 3.8. Semantic Correctness Evaluation

Existing studies on LLM-based test generation primarily evaluate syntactic correctness, code coverage, or executability of generated tests. However, these metrics alone do not fully capture whether generated tests semantically reflect the intent of the original natural language requirements. Since the proposed approach focuses on transforming requirements into executable system level tests, an additional semantic correctness evaluation was introduced in this work. The purpose of this evaluation was to assess how accurately generated tests correspond to the expected behaviour described in the original requirements, including endpoint usage, request structure, assertions, and expected responses.

In this research, semantic correctness refers to the degree to which the generated pytest test bundle validates the intended behaviour described in the originating natural language requirement. The evaluation focuses on whether the generated tests meaningfully verify the expected system

behaviour, assertions, authorization handling, validation logic, and error conditions associated with the requirement.

The proposed prototype generates one pytest file for each requirement. Each generated pytest file may contain multiple pytest test functions representing positive scenarios, negative scenarios, authorization checks, boundary conditions, or alternative execution paths related to the same requirement. Since a single test function may only partially validate a requirement, semantic correctness was evaluated at the requirement file level rather than at the individual test function level. This approach reflects the practical nature of system testing, where multiple related test cases are often required to sufficiently validate one functional requirement.

Semantic correctness was evaluated manually by the author acting as a human evaluator with software testing knowledge. Each generated pytest file was compared against its originating requirement and assigned one of three semantic correctness scores shown in Table 7.

| **Score** | **Description** |
|---|---|
| 0 | Generated tests do not meaningfully validate the intended requirement behaviour |
| 1 | Generated tests partially validate the requirement but miss important behaviours, assertions, or execution paths |
| 2 | Generated tests sufficiently validate the intended requirement behaviour |

Table 7. Test quality scoring criteria

The overall semantic correctness score was calculated using the following formula:

$$SC = \frac{\sum RS}{2N}$$

where:

- $SC$ — Semantic Correctness
- $RS$ — Requirement Score
- $N$ — Number of Requirements

The resulting value is normalized to the range from 0 to 1, where 0 represents completely incorrect semantic alignment and 1 represents fully correct semantic alignment for all evaluated requirements.

To support interpretation of the obtained scores, the following qualitative interpretation scale was used:

- 0.90–1.00 : Highly aligned with requirement intent
- 0.70–0.89 : Mostly aligned with minor omissions
- 0.50–0.69 : Partially aligned
- Below 0.50 : Weak alignment with requirement intent

Although the evaluation remains partially subjective due to manual assessment, the scale provides a structured way to estimate how accurately generated executable tests reflect the semantics of the original requirements. Nevertheless, the evaluation provides an additional qualitative perspective complementing executability and coverage based metrics used throughout the experimental analysis.

Table 8 presents the semantic correctness evaluation results for both evaluated APIs.

| **API** | **Req.** | **Correct (2)** | **Partial (1)** | **Incorrect (0)** | **SC** |
|---|---|---|---|---|---|
| Fixer API | 37 | 25 | 12 | 0 | 0.838 |
| PayPal Checkout Orders API | 30 | 13 | 17 | 0 | 0.717 |
| **Total** | **67** | **38** | **29** | **0** | **0.784** |

Table 8. Semantic correctness evaluation results

The evaluation results show that all generated requirement level pytest files were at least partially semantically aligned with their originating requirements. No generated requirement level test bundle was classified as semantically incorrect. Overall, 38 out of 67 generated pytest files were evaluated as fully correct, while 29 were classified as partially correct.

The Fixer API achieved a higher semantic correctness score than the PayPal Checkout Orders API. This difference is mainly explained by the simpler and more stateless structure of the Fixer API, which primarily consists of read only GET operations with predictable responses. In contrast, the PayPal Checkout Orders API contains more complex workflows involving authorization handling, order state transitions, approval flows, tracking operations, and environment dependent execution paths.

Most partially correct evaluations were caused not by unrelated or hallucinated test generation, but by external system constraints and limitations of live API execution environments. Common causes included subscription plan restrictions, authorization requirements, state dependent workflows requiring manual approval, unavailable runtime states, and conditional execution paths that prevented full validation of some scenarios. In several cases, generated tests validated related operations or generalized authorization behaviour rather than the exact endpoint specific behaviour described in the requirement.

Direct comparison with existing studies remains difficult because most previous research primarily evaluates executability, code coverage, or syntactic correctness rather than semantic alignment between requirements and executable tests. Nevertheless, the obtained results suggest that the proposed approach is capable of generating executable tests that preserve substantial parts of the behavioural intent described in natural language requirements.

The semantic correctness evaluation additionally demonstrated that higher quality results were generally achieved for simpler and more stateless APIs, while more complex APIs involving authorization flows, state transitions, and environment dependent execution paths introduced additional challenges for accurate requirement level validation.

## 3.9. Efficiency Evaluation

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.

Timing measurements were collected for complete test execution runs, including the generation of pytest test cases, validation of generated code, and execution of the tests against the target API. Descriptive statistics for execution time are presented in Table 9, including average, median, minimum, and maximum values.

| API | Avg (s) | Median (s) | Min (s) | Max (s) |
|---|---|---|---|---|
| Fixer API | 27.89 | 27.10 | 12.99 | 44.81 |
| PayPal Checkout Orders API | 32.84 | 31.43 | 17.10 | 46.08 |

Table 9. Test generation time summary statistics across APIs based on requirement

For the Fixer API, the average execution time was 27.89 seconds, with a median of 27.10 seconds, a minimum of 12.99 seconds, and a maximum of 44.81 seconds. The relatively small difference between average and median values indicates stable execution behaviour, suggesting that test generation and execution times are relatively consistent across different requirements. Variability in execution time can be attributed to differences in the number of generated test cases per requirement and the presence of retry attempts following validation failures.

For the PayPal Checkout Orders API, execution times were higher, with an average of 32.84 seconds and a median of 31.43 seconds. The minimum and maximum execution times were 17.10 seconds and 46.08 seconds, respectively. The increased execution time reflects the higher complexity of the PayPal API, which involves state dependent workflows, more complex authorization mechanisms, and additional validation logic. These characteristics result in longer execution times, particularly when retries are triggered due to authorization failures or unexpected response formats.

Comparing the two APIs, the difference in execution time demonstrates that the proposed approach scales reasonably with increasing system complexity. Although the PayPal API introduces more complex workflows and stricter execution constraints, the overall execution time remains within a similar range to that of the simpler Fixer API. This suggests that the approach can be applied to APIs of varying complexity without excessive performance degradation.

In addition to automated execution measurements, a limited practical comparison with manually implemented tests was performed in order to provide additional context regarding development effort associated with requirement level REST API test creation. The author manually implemented several requirement level pytest test files for selected REST API requirements without using automated test generation.

The first manually created pytest test file required approximately 12 minutes to implement. This initial implementation included preparation of the project structure, configuration of HTTP request handling, authentication setup, creation of assertions, and organization of the pytest testing structure. After the initial setup phase, subsequent manually created requirement level pytest test

files required significantly less effort due to reuse of the previously implemented structure and helper components. The average implementation time for additional manually written requirement level pytest files was approximately 4 minutes per requirement.

| **Activity** | **Manual Implementation** | **Proposed Approach** |
|---|---|---|
| Pytest structure creation | Manual | Automatic |
| HTTP request construction | Manual | Automatic |
| Assertion implementation | Manual | Automatic |
| Authorization handling setup | Manual | Automatic |
| Average implementation effort per requirement | Approximately 4 minutes | Tens of seconds |

Table 10. Limited comparison between manual and automated test generation

During manual implementation, the primary effort consisted of adapting request payloads, assertions, endpoint specific validation logic, authorization handling, expected status code verification, and negative scenario checks according to individual requirement logic.

The comparison remains limited because manual implementation effort strongly depends on developer experience, familiarity with the target API, and requirement complexity. Therefore, the presented comparison should be interpreted as an illustrative efficiency comparison rather than a rigorous productivity benchmark.

Although this comparison does not represent a large scale controlled experiment, it provides practical insight into the amount of manual effort required for requirement driven REST API system test implementation. The observations suggest that the proposed approach can reduce repetitive implementation effort associated with requirement level REST API test creation, particularly for creation of initial test structures, standardized request handling, and repetitive validation logic.

Overall, the efficiency results indicate that the proposed approach enables rapid generation and execution of system level tests. Even for complex APIs, complete test runs can be performed within tens of seconds, supporting the suitability of the approach for iterative testing and continuous integration scenarios.

## 3.10. Comparative Analysis

This section situates the proposed approach within the broader landscape of automated test generation techniques. The comparison is qualitative and analytical, focusing on conceptual design choices, input artifacts, automation level, and practical applicability. Direct performance benchmarking was not conducted due to differences in execution environments, testing objectives, and required system access among the evaluated approaches.

### 3.10.1. Comparison with CiRA

CiRA [FFV21] focuses on extracting conditional logic from natural language requirements to derive acceptance tests. Its primary goal is to improve requirement coverage and support human

testers during test design. The generated outputs are abstract acceptance test cases rather than executable test scripts.

In contrast, the proposed approach generates fully executable system level API tests. While CiRA emphasizes logical completeness of requirements, it does not address runtime execution, interaction with deployed systems, or validation against real API behaviour. The proposed approach complements CiRA by extending requirement based testing to black-box system validation against external APIs.

### 3.10.2. Comparison with APITestGenie

APITestGenie [PLF24] employs large language models to generate API test scripts from business requirements and OpenAPI specifications. The approach demonstrates that LLMs can assist in producing executable test cases and reduce manual test design effort. However, the evaluation primarily focuses on test generation success and qualitative inspection of generated scripts.

A key difference between APITestGenie[PLF24] and the proposed approach lies in result validation and execution analysis. APITestGenie does not define a structured validation framework for assessing execution outcomes, such as systematic classification of failures, execution stability, or endpoint level coverage. Generated tests are typically evaluated based on generation success or basic execution feedback, without deeper analysis of runtime behaviour.

In addition, the experimental evaluation of APITestGenie [PLF24] relies largely on simplified benchmark APIs, such as Petstore like services. While such APIs are commonly used for illustrative purposes, they exhibit limited authorization logic, minimal state dependency, and stable response behaviour. As a result, they do not fully reflect the complexity of real world external APIs.

In contrast, the proposed approach explicitly evaluates generated tests through execution against production grade external APIs with realistic constraints, including authentication, subscription plans, state dependent workflows, and inconsistent error semantics. This enables systematic assessment of execution outcomes, endpoint coverage, execution stability, and failure diagnostic value.

Consequently, while APITestGenie [PLF24] demonstrates the feasibility of LLM assisted API test generation, the proposed approach extends this line of work by introducing structured runtime validation and by evaluating test behaviour in more challenging and realistic testing environments.

### 3.10.3. Comparison with LLM Testing Frameworks

Frameworks such as CCTest [LWL+23] and CODET [CZN+23] focus on testing and improving the reliability of LLM generated code by detecting inconsistencies across multiple generations. These approaches treat the LLM itself as the system under test.

In contrast, the proposed approach treats the LLM as a test generation component and evaluates the behaviour of the external system under test. While testing frameworks address LLM correctness, the proposed approach addresses system level validation and requirement conformance, operating at a different abstraction level.

### 3.10.4. Comparison with "Open API Testing Using Large Language Models and Prompt Engineering"

The work presented in "Open API Testing Using Large Language Models and Prompt Engineering" [KLB25] investigates the use of large language models in combination with prompt design strategies to generate API test cases from structured API descriptions and prompt templates. This research emphasizes prompt engineering techniques to guide the LLM toward generating a diverse set of API tests, often focusing on exploiting the expressiveness of prompts to cover conditional logic, edge cases, or scenario variations.

While this approach and the proposed work both rely on large language models for test generation, there are key differences in goals and evaluation focus. The prompt engineering work generally assumes OpenAPI specifications or API documentation as the primary input and investigates how variations in prompt structure affect the richness and diversity of generated test cases. The emphasis is on improving test diversity and discovering broader behavioural scenarios based on prompt heuristics.

In contrast, the proposed approach treats natural language functional requirements as the primary driver of test generation, using OpenAPI context to ground the requirement interpretation. The output of the proposed system is a suite of **fully executable pytest tests**, which are automatically validated for syntactic correctness and executed against live external APIs without manual corrections. The evaluation of the proposed approach explicitly measures execution outcomes, endpoint coverage, execution stability, and failure diagnostic value, whereas the prompt engineering work generally evaluates generation quality or diversity without full system execution.

While both approaches leverage LLM capabilities, the proposed approach prioritizes requirement traceability and practical test execution as first class evaluation criteria, extending beyond prompt engineering techniques to demonstrate end-to-end test generation and validation.

| **Aspect** | **Proposed** | **CiRA** | **APITestGenie** | **Prompt Engineering Paper** |
|---|---|---|---|---|
| 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 |
| External API testing | Yes | No | Limited | Not evaluated |
| Execution stability evaluated | Yes | No | No | Not evaluated |
| Failure diagnostics | High | Medium | Medium | Not evaluated |

Table 11. Extended comparison with related work

## 3.11. Observed Limitations

Analysis of failed test cases revealed several recurring failure categories:

1. Specification mismatch The OpenAPI specification did not always accurately reflect runtime behaviour, particularly regarding HTTP status codes and error semantics.
2. Subscription plan limitations Certain endpoints, such as currency conversion and base

currency changes, returned forbidden responses despite valid input, due to subscription restrictions.

3. Ambiguous error handling Invalid input often resulted in API specific error codes rather than standard HTTP error responses.
4. Non deterministic response formats Some endpoints returned non JSON responses (e.g., empty responses or JSONP), causing parsing failures.
5. External execution constraints Some operations could not be executed due to sandbox or permission restrictions and were therefore skipped.
6. State dependent behaviour APIs with multi step workflows (e.g., order approval and capture) require additional context beyond individual requirements. Missing state information led to assertion failures.

These failure categories indicate limitations not in test generation itself, but in the predictability and consistency of the tested API.

# 4. Conclusions

## 4.1. Main Findings

This thesis investigated the feasibility of generating executable system level REST API tests directly from natural language requirements using large language models and OpenAPI specifications. The proposed approach was designed for black-box testing scenarios, where internal system details are unavailable and testing must rely solely on externally observable behaviour. The main contribution of this work is a prototype approach for transforming natural language requirements and OpenAPI specifications into executable pytest based REST API system tests using large language models together with a semantic correctness evaluation methodology for requirement level validation.

The experimental evaluation demonstrated that the approach can reliably generate syntactically valid and executable pytest based test cases without manual intervention. For both evaluated APIs, tests were successfully generated for all evaluated requirements, ensuring complete requirement–test traceability at the generation level. Endpoint reachability coverage reached 100% for both APIs, indicating that the approach is capable of exercising the full API surface defined by the OpenAPI specifications.

In addition to executability and endpoint coverage evaluation, semantic correctness analysis was performed at the requirement file level using manual expert evaluation and a predefined scoring rubric. The results showed that the generated test bundles were generally well aligned with the intended requirement behaviour, achieving an overall normalized semantic correctness score of 0.784. Most partially correct evaluations were caused by external execution constraints, state dependent workflows, authorization requirements, or environment limitations rather than unrelated or semantically incorrect test generation.

Execution stability analysis showed that all generated tests behaved deterministically across repeated executions. When test isolation was maintained by renewing authorization state where required, no test exhibited flaky behaviour. This indicates that the generated test logic itself is stable and that observed execution failures are reproducible and attributable to external system conditions rather than nondeterministic test generation.

A practical comparison with manually created tests provided additional insight into the amount of effort required for requirement level REST API test implementation. The first manually implemented pytest test file required approximately 12 minutes to create due to initial project setup, request handling configuration, and assertion preparation. Subsequent manually written requirement level pytest files required approximately 4 minutes on average, primarily because previously implemented structures and helper logic could be reused.

In comparison, the proposed automated approach generated, validated, and executed requirement level test bundles within tens of seconds on average. These observations indicate that the proposed approach can substantially reduce repetitive implementation effort associated with requirement level REST API test creation.

The qualitative comparison with existing approaches from the literature highlights the distinct

position of the proposed approach. While methods such as CiRA focus on generating abstract acceptance tests and prompt engineering based approaches emphasize test diversity through prompt design, the proposed approach prioritizes end-to-end test executability, runtime validation, and requirement level traceability. Unlike LLM testing frameworks such as CCTest, which evaluate the reliability of LLM outputs, this work evaluates the behaviour of real systems using LLM generated tests. As a result, the approach complements existing research by bridging the gap between requirement interpretation and executable system level validation.

Overall, the results indicate that large language models can be effectively leveraged to automate generation of executable REST API tests from natural language requirements, provided that execution constraints and external system dependencies are carefully considered. The proposed approach is particularly well suited for validating external APIs in black-box environments, where traditional white-box testing techniques are not applicable.

## 4.2. Limitations

Several limitations of the proposed approach were identified during experimental evaluation.

A stricter analysis of complete endpoint test coverage revealed that not all endpoints achieved full execution success across all generated tests. These limitations were primarily caused by external factors, including authorization constraints, subscription plan restrictions, sandbox limitations, and undocumented or inconsistent API behaviour. Importantly, these factors affected test execution outcomes rather than the ability of the approach to generate meaningful tests.

The manual comparison performed in this work does not represent a controlled experimental study and should therefore be interpreted as an illustrative efficiency comparison rather than a rigorous productivity benchmark. Nevertheless, the observations suggest that automated generation can substantially reduce repetitive implementation effort and accelerate creation of executable REST API system tests.

Another limitation of this work is that the natural language requirements used during experimentation were manually created by the author from OpenAPI specifications rather than collected from independent stakeholders. Although this approach ensured consistency of evaluation, it may not fully reflect the variability, ambiguity, and inconsistency commonly present in real industrial requirements documentation.

## 4.3. Future Work

Future work may extend this approach by incorporating explicit workflow models to improve handling of state dependent requirements, enhancing assertion adaptation based on observed API behaviour, and exploring hybrid evaluation strategies that combine qualitative execution analysis with quantitative fault detection metrics. Additional research could also investigate retrieval-augmented generation techniques, automated oracle generation, and integration of the approach into continuous integration environments. Future evaluation should additionally include larger and independently authored industrial REST API requirement datasets in order to assess

the robustness of the proposed approach under realistic requirement ambiguity and evolving stakeholder specifications.

# References


[AB23] C. Antoniou, N. Bassiliades. A tool for requirements engineering using ontologies and boilerplates. *Automated Software Engineering*. 2023, volume 31, number 1, p. 5. ISSN 1573-7535. Available from: https://doi.org/10.1007/s10515-023-00403-y. Published on November 21, 2023.

[Abd20] S. Abdullahi. Software testing: review on tool, techniques and challenges. *International Journal of Advanced Research in Technology and Innovation*. 2020. Available also from: https://www.researchgate.net/publication/342391520_Software_testing_review_on_tool_techniques_and_challenges.

[AIA+19] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan, I. Porres. Model-based testing using UML activity diagrams: A systematic mapping study. *Computer Science Review*. 2019, volume 33, pp. 98–112. ISSN 1574-0137. Available from: https://doi.org/https://doi.org/10.1016/j.cosrev.2019.07.001.

[ATT+25] S. Alagarsamy, C. Tantithamthavorn, W. Takerngsaksiri, C. Arora, A. Aleti. Enhancing large language models for text-to-testcase generation. *Journal of Systems and Software*. 2025, volume 230, p. 112531. ISSN 0164-1212. Available from: https://doi.org/https://doi.org/10.1016/j.jss.2025.112531.

[BFG+21] O. Baniaș, D. Florea, R. Gyalai, D.-I. Curiac. Automated Specification-Based Testing of REST APIs. *Sensors*. 2021, volume 21, number 16. ISSN 1424-8220. Available from: https://doi.org/10.3390/s21165375.

[CCV+21] S. Casas, D. Cruz, G. Vidal, M. Constanzo. Uses and applications of the OpenAPI/Swagger specification: a systematic mapping of the literature. In: *2021 40th International Conference of the Chilean Computer Science Society (SCCC)*. 2021. Available from: https://doi.org/10.1109/SCCC54552.2021.9650408.

[CZN+23] 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. Available also from: https://openreview.net/forum?id=ktrw68Cmu9c.

[DF14] 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.

[Don24] C. Dongmo. A Review of Non-Functional Requirements Analysis Throughout the SDLC. *Computers*. 2024, volume 13, number 12. ISSN 2073-431X. Available from: https://doi.org/10.3390/computers13120308.

[EAC+22] A. Ehsan, M. A. M. E. Abuhaliqa, C. Catal, D. Mishra. RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions. *Applied Sciences*. 2022, volume 12, number 9. ISSN 2076-3417. Available from: https://doi.org/10.3390/app12094369.

[FFV+23] 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 0164-1212. Available from: https://doi.org/https://doi.org/10.1016/j.jss.2022.111549.

[FFV21] J. Fischbach, J. Frattini, A. Vogelsang. *CiRA: A Tool for the Automatic Detection of Causal Relationships in Requirements Artifacts*. 2021. Available also from: https://arxiv.org/abs/2103.06768.

[Fie00] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[FPQ+23] X. Franch, C. Palomares, C. Quer, P. Chatzipetrou, T. Gorschek. The state-of-practice in requirements specification: an extended interview study at 12 companies. *Requirements Engineering*. 2023, volume 28, number 3, pp. 377–409. ISSN 1432-010X. Available from: https://doi.org/10.1007/s00766-023-00399-7.

[Gla11] L. Glaves. API design for C++ by Martin Reddy. *SIGSOFT Softw. Eng. Notes*. 2011. ISSN 0163-5948. Available from: https://doi.org/10.1145/2020976.2021002.

[GZA23] A. Golmohammadi, M. Zhang, A. Arcuri. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 2023, volume 33, number 1. ISSN 1049-331X. Available from: https://doi.org/10.1145/3617175.

[HEJ+21] R. Haas, D. Elsner, E. Juergens, A. Pretschner, S. Apel. How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens, Greece: Association for Computing Machinery, 2021, pp. 1281–1291. ESEC/FSE 2021. ISBN 9781450385626. Available from: https://doi.org/10.1145/3468264.3473922.

[HR25] M. Hamburg, A. Roman. Black-Box Testing for Practitioners: A Case of the New ISTQB Test Analyst Syllabus. In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2025, pp. 634–645. Available from: https://doi.org/10.1109/ICST62969.2025.10988991.

[YFY23] J. Yoon, R. Feldt, S. Yoo. *Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing*. 2023. Available also from: https://arxiv.org/abs/2311.08649.

[YHC+25] Z. Yang, R. Huang, C. Cui, N. Niu, D. Towey. *Requirements-Based Test Generation: A Comprehensive Survey*. 2025. Available also from: https://arxiv.org/abs/2505.02015.

[Jia23] X. Jia. The Role and Importance of Software Testing in Software Quality Management. *Journal of Industry and Engineering Management*. 2023. Available from: https://doi.org/10.62517/jiem.202303406.

[KLB25] 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. Available also from: https://api.semanticscholar.org/CorpusID:284157523.

[KT24] H. Kirinuki, H. Tanno. *ChatGPT and Human Synergy in Black-Box Testing: A Comparative Analysis*. 2024. Available also from: https://arxiv.org/abs/2401.13924.

[LCS+24] 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 NLP-based unified boilerplate approach. *Journal of Systems and Software*. 2024, volume 211. ISSN 0164-1212. Available from: https://doi.org/https://doi.org/10.1016/j.jss.2024.112005.

[LF22] S. Lukasczyk, G. Fraser. Pynguin: automated unit test generation for Python. In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 168–172. ISBN 9781450392235. Available from: https://doi.org/10.1145/3510454.3516829.

[LWL+23] Z. Li, C. Wang, Z. Liu, H. Wang, D. Chen, S. Wang, C. Gao. *CCTEST: Testing and Repairing Code Completion Systems*. 2023. Available also from: https://arxiv.org/abs/2208.08289.

[Mas25] A. Maspupah. LITERATURE REVIEW: ADVANTAGES AND DISADVANTAGES OF BLACK BOX AND WHITE BOX TESTING METHODS. *Jurnal Techno Nusa Mandiri*. 2025, volume 21, number 2, pp. 151–162. Available from: https://doi.org/10.33480/techno.v21i2.5776.

[MCS+24] M. Mirchev, A. Costea, A. K. Singh, A. Roychoudhury. *Assured Automatic Programming via Large Language Models*. 2024. Available also from: https://arxiv.org/abs/2410.18494.

[NDG24] S.-C. Necula, F. Dumitriu, V. Greavu-Șerban. A Systematic Literature Review on Using Natural Language Processing in Software Requirements Engineering. *Electronics*. 2024, volume 13, number 11. ISSN 2079-9292. Available from: https://doi.org/10.3390/electronics13112055.

[PLF24] 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.

[SNE+24] 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. Available from: https://doi.org/10.1109/TSE.2023.3334955.

[STS+23] B. Steenhoek, M. Tufano, N. Sundaresan, A. Svyatkovskiy. *Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation*. 2023. Available also from: https://arxiv.org/abs/2310.02368.

[TJT+25] T. Tahir, H. Jahankhani, K. Tasleem, B. Hassan. Cross-Project Multiclass Classification of EARS-Based Functional Requirements Utilizing Natural Language Processing, Machine Learning, and Deep Learning. *Systems*. 2025, volume 13, number 7. ISSN 2079-8954. Available from: https://doi.org/10.3390/systems13070567.

[VF25] A. Vogelsang, J. Fischbach. Using Large Language Models for Natural Language Processing Tasks in Requirements Engineering: A Systematic Guideline. In: *Handbook on Natural Language Processing for Requirements Engineering*. Edited by A. Ferrari, G. Ginde. Cham: Springer Nature Switzerland, 2025, pp. 435–456. Available from: https://doi.org/10.1007/978-3-031-73143-3_16.

[WHC+24] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, Q. Wang. *Software Testing with Large Language Models: Survey, Landscape, and Vision*. 2024. Available also from: https://arxiv.org/abs/2307.07221.

# Abbreviations

AI – artificial intelligence
NLP – natural language processing
LLM – large language model