UNIVERSIDADE FEDERAL FLUMINENSE

THIAGO DA SILVA BARRADAS

Combining TSL and LLM to Automate REST API Testing: Approach RestTSLLM

NITERÓI

2026

THIAGO DA SILVA BARRADAS

Combining TSL and LLM to Automate REST API Testing: Approach RestTSLLM

Master's Dissertation presented to the Computing Postgraduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master in Computing. Concentration area: Computer Science.

Advisor:
VÂNIA DE OLIVEIRA NEVES

NITERÓI

2026

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

B268c Barradas, Thiago da Silva
Combining TSL and LLM to Automate REST API Testing: Approach RestTSLLM / Thiago da Silva Barradas. - 2026.
176 f.: il.

Orientador: Vânia de Oliveira Neves.
Dissertação (mestrado)-Universidade Federal Fluminense, Instituto de Computação, Niterói, 2026.

1. Engenharia de software. 2. Teste (Computação). 3. Inteligência artificial. 4. Produção intelectual. I. Neves, Vânia de Oliveira, orientadora. II. Universidade Federal Fluminense. Instituto de Computação. III. Título.

CDD - XXX

Bibliotecário responsável: Debora do Nascimento - CRB7/6368

THIAGO DA SILVA BARRADAS

COMBINING TSL AND LLM TO AUTOMATE REST API TESTING: APPROACH RESTTSLLM

Master's Dissertation presented to the Computing Postgraduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master in Computing. Concentration area: Computer Science.

Approved in July 2026.

EXAMINATION BOARD

Prof. Vânia de Oliveira Neves - Advisor, UFF

Prof. Aline Marins Paes Carvalho, UFF

Prof. Andre Takeshi Endo, UFSCAR

Niterói

2026

All you touch and all you see is all your life will ever be. (Pink Floyd, Breathe, 1973)

Acknowledgements

I would like to thank everyone who supported me throughout this project. First and foremost, I am grateful to my family: my wife Lorena, my parents Mena and Marcelino, and my siblings Raphael and Cristiane, who helped me in so many ways that an entire book would not be enough to describe them.

My deepest gratitude goes to my advisor, for all the trust placed in me and in this project, and for all the guidance along the way. I also thank all my friends, especially the Nikit family, and particularly Matheus Moreira - wherever he may be - who has always been a great friend, brother, mentor, and a source of motivation for my achievements.

I am also grateful to all the professors at the Institute of Computing who guided me on this journey of learning.

Finally, to everyone who, in one way or another, contributed to making this dream come true.

Abstract

The effective execution of tests for REST APIs remains a considerable challenge for development teams, driven by the inherent complexity of distributed systems, the multitude of possible scenarios, and the limited time available for test design. Exhaustive testing of all input combinations is impractical, often resulting in undetected failures, high manual effort, and limited test coverage. To address these issues, we introduce RestTSLLM, an approach that uses Test Specification Language (TSL) in conjunction with Large Language Models (LLMs) to automate the generation of test cases for REST APIs. The approach targets two core challenges: the creation of test scenarios and the definition of appropriate input data. The proposed solution integrates prompt engineering techniques with an automated pipeline to evaluate various LLMs on their ability to generate tests from OpenAPI Specification (OAS). The evaluation focused on metrics such as success rate, test coverage, and mutation score, enabling a systematic comparison of model performance. The results indicate that the best-performing LLMs – Claude 3.5 Sonnet (Anthropic), Deepseek R1 (Deepseek), Qwen 2.5 32b (Alibaba), and Sabiá 3 (Maritaca) – consistently produced robust and contextually coherent REST API tests. Among them, Claude 3.5 Sonnet outperformed all other models across every metric, emerging in this study as the most suitable model for this task. These findings highlight the potential of LLMs to automate the generation of tests based on API specifications. To complement this technical evaluation, we conducted an acceptance survey with 96 practitioners involved in the REST API development lifecycle. The results revealed a high level of acceptance of the approach (overall agreement mean of 4.28 on a 1–5 scale), with practitioners particularly valuing its perceived usefulness and the intermediate TSL representation, while pointing to trust in the generated tests and human review as the main considerations for adoption.

Keywords: Test Automation, Large Language Models, Integration Testing, REST API Testing, AI in Software Testing, Test Generation.

Resumo

A execução eficaz de testes para REST APIs continua sendo um desafio considerável para as equipes de desenvolvimento, impulsionado pela complexidade inerente dos sistemas distribuídos, pela multiplicidade de cenários possíveis e pelo tempo limitado disponível para o design dos testes. A testagem exaustiva de todas as combinações de entrada é impraticável, resultando frequentemente em falhas não detectadas, elevado esforço manual e cobertura de testes limitada. Para enfrentar esses problemas, apresentamos a RestTSLLM, uma abordagem que utiliza a Test Specification Language (TSL) em conjunto com Large Language Models (LLMs) para automatizar a geração de casos de teste para REST APIs. A abordagem define dois desafios centrais: a criação de cenários de teste e a definição de dados de entrada apropriados. A solução proposta integra técnicas de prompt engineering a um fluxo automatizado para avaliar diversos LLMs em sua capacidade de gerar testes a partir de especificações OpenAPI (OAS). A avaliação concentrou-se em métricas como taxa de sucesso, cobertura de testes e pontuação de mutação, possibilitando uma comparação sistemática do desempenho dos modelos. Os resultados indicam que os LLMs com melhor desempenho – Claude 3.5 Sonnet (Anthropic), Deepseek R1 (Deepseek), Qwen 2.5 32b (Alibaba), and Sabiá 3 (Maritaca) – produziram de forma consistente testes robustos e contextualmente coerentes para REST APIs. Entre eles, o Claude 3.5 Sonnet superou todos os outros modelos em todas as métricas, destacando-se neste estudo como o modelo mais adequado para essa tarefa. Esses achados ressaltam o potencial dos LLMs para automatizar a geração de testes com base em especificações de APIs. Para complementar essa avaliação técnica, conduzimos um survey de aceitação com 96 profissionais envolvidos no ciclo de desenvolvimento de REST APIs. Os resultados revelaram um alto nível de aceitação da abordagem (média geral de concordância de 4,28 em uma escala de 1 a 5), com os profissionais valorizando especialmente sua utilidade percebida e a representação intermediária em TSL, ao mesmo tempo em que apontaram a confiança nos testes gerados e a revisão humana como as principais considerações para adoção.

Palavras-chave: Automação de Testes, Large Language Models, Testes de Integração, Testes de REST API, IA em Testes de Software, Geração de Testes.

List of Figures

List of Tables

List of Listings

List of Abbreviations and Acronyms

AAA Arrange, Act, Assert

AI Artificial Intelligence

API Application Programming Interface

BDD Behavior-Driven Development

CD Continuous Delivery

CI Continuous Integration

CLI Command Line Interface

CLOC Count Lines of Code

CRUD Create, Read, Update, Delete

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JSON JavaScript Object Notation

LLM Large Language Model

MIT Massachusetts Institute of Technology (License)

ML Machine Learning

NLP Natural Language Processing

OAS OpenAPI Specification

QA Quality Assurance

RAG Retrieval-Augmented Generation

REST Representational State Transfer

RQ Research Question

SE Software Engineering

SQL Structured Query Language

SUT System Under Test

TDD Test-Driven Development

TSL Test Specification Language

URL Uniform Resource Locator

XML Extensible Markup Language

Contents

1 Introduction

1.1 Software Testing: Importance and Challenges

Software testing is an essential component of the system development lifecycle, playing a crucial role in ensuring the quality and reliability of systems (TUTEJA; DUBEY, et al., 2012; JINDAL, 2016; ANAND; UDDIN, 2019). Over the years, the increasing complexity of computational systems and the consequences of undetected failures have highlighted the need for robust and well-structured testing practices at all stages of development (TUTEJA; DUBEY, et al., 2012; JINDAL, 2016; ANAND; UDDIN, 2019).

Despite its importance, the effective execution of tests is not trivial. Teams often face challenges due to the complexity of systems, the volume of tests, and the limited time available for testing design and execution (TUTEJA; DUBEY, et al., 2012; NEWMAN, 2015; MENDOZA et al., 2024). Exhaustively testing all possible inputs is unfeasible, especially in systems that require high reliability, which consumes a significant portion of the development effort (TUTEJA; DUBEY, et al., 2012). Additionally, manual test execution can result in incomplete scenarios and difficulty in detecting subtle failures (TUTEJA; DUBEY, et al., 2012), leading to high costs and limited coverage (JINDAL, 2016).

Several studies have contributed to advancing software testing techniques, particularly by addressing challenges such as test input generation and the complexities inherent in modern real-world systems, and remain open problems (ORSO; ROTHERMEL, 2014). Much progress is still needed to fully address all these difficulties, particularly in improving the effectiveness of generated tests (i.e., producing meaningful test scenarios) and automating testing processes end-to-end.

Recent studies have increasingly explored the use of Large Language Models (LLMs)—neural network models trained on large text corpora that learn to predict and generate natural language, and that acquire broad capabilities across tasks without task-specific

retraining (HOU et al., 2024; HAGOS; BATTLE; RAWAT, 2024)—as a way to accelerate and improve software testing activities (BOUKHLIF; KHARMOUM; HANINE, 2024). LLMs have rapidly and consistently gained prominence in task automation and are increasingly recognized as promising tools to tackle many of the key challenges in software engineering in the coming years (PEZZÈ et al., 2024). These include reducing the costs associated with test case generation and execution, reducing the overall complexity of testing activities and improving the contextual relevance of automatically generated tests (PEZZÈ et al., 2024; BOUKHLIF; KHARMOUM; HANINE, 2024).

1.2 REST APIs and the Role of Integration Testing

Modern and integrated system architectures widely adopt the REST API (Representational State Transfer, Application Programming Interface) model in software development due to its simplicity, flexibility, and scalability (FIELDING, 2000; LERCHER, 2024). The REST API has become the standard for communication between different services, providing a uniform and lightweight way to integrate heterogeneous systems through HTTP communication. One of the main advantages of REST is its compatibility with the web and its ability to handle large volumes of distributed transactions, where communication between independent components is essential to maintain modularity and scalability (FIELDING, 2000; LERCHER, 2024).

Representational State Transfer (REST) was formally defined by Fielding (2000) as an architectural style for distributed hypermedia systems. In this style, a *resource* is any named concept accessible via the web—a user, a product, an order—identified by a Uniform Resource Locator (URL); a *representation* is the data encoding of that resource's current state exchanged between client and server, typically in JavaScript Object Notation (JSON). A REST API exposes these resources through a uniform interface built on top of the HTTP protocol, enabling interoperability between heterogeneous systems without coupling client and server implementations (FIELDING, 2000; LERCHER, 2024).

With the widespread adoption of REST APIs, integration testing is a crucial strategy for improving software quality because it ensures that different system components interact correctly, offering a more realistic view of how an application behaves in a production environment (DELAMARO; JINO; MALDONADO, 2013; SOMMERVILLE, 2016; SOTIRIADIS et al., 2017; GOLMOHAMMADI; ZHANG; ARCURI, 2023; NEWMAN, 2015). Unlike unit tests, which target isolated functionalities, integration tests focus on interacti-

ons between services, modules, or systems, ensuring the correct exchange of information, such as inputs, responses, and communication flows. According to Newman (2015), failing to ensure proper integration between components can result in critical communication issues, such as inadequate responses.

Modern systems are increasingly built as microservices that interact through REST APIs, where a single client request may involve a sequence of calls across multiple services (CAO; PANICHELLA; VERWER, 2025). Consequently, a significant portion of system behavior emerges from these service interactions, and testing approaches focused solely on unit-level objectives may fail to capture complex system-level behaviors and the faults arising from service dependencies (CAO; PANICHELLA; VERWER, 2025). The complexity of microservice applications therefore requires automated testing strategies that focus on service interactions, making integration and API-level testing fundamental for validating REST API behavior in realistic environments (REILE et al., 2022).

Recent studies highlight the increasing attention to REST API testing, with emphasis on automation techniques, schema-based fuzzing, and test coverage metrics (GOLMOHAMMADI; ZHANG; ARCURI, 2023). Golmohammadi, Zhang, and Arcuri (2023) identified black-box testing approaches and schema-guided fuzzers as the most common strategies, while also pointing out challenges like oracle definition and handling of real-world scenarios. Other works (OLSTHOORN, 2022) reinforce the importance of integration testing and the difficulties in generating valid inputs for complex interaction flows.

Creating such tests directly depends on the mapped scenarios and their respective input data. This process still faces several challenges, mainly due to the inherent complexity of interactions between system components, such as services or databases. A major obstacle lies in ensuring that inputs are both valid, and coherent with each component's constraints, which requires a deep understanding of business rules, and system contracts. Moreover, automatic input generation methods often fail to adequately exercise all critical integration paths (OLSTHOORN, 2022; GOLMOHAMMADI; ZHANG; ARCURI, 2023).

1.3 Motivation and Research Gaps

A recent survey by Wang et al. (2024) reviews 102 studies on the application of LLMs in supporting various software testing tasks. In particular, LLMs have shown promise in generating test inputs, test oracles, and behavior-based tests–especially in contexts where textual descriptions are transformed into structured test cases–a key aspect in scenarios

involving REST APIs, which often rely on standardized documentation such as OpenAPI Specification (OAS) [1]. Despite the growing interest in applying LLMs to software testing, Wang et al. (2024) also identify topics that remain underexplored. This is particularly true for more complex testing scenarios, such as those involving integrated systems and components.

Complementing the survey by Wang et al. (2024), the study by Alshahwan, Harman, and Marginean (2023) extends these findings from an industrial perspective, emphasizing the importance of advancements in test automation, especially in complex environments such as the integration of distributed systems and REST APIs. Their study highlights challenges in test data generation, maintenance cost reduction, and improved test coverage– areas that can benefit from using Artificial Intelligence (AI) techniques such as LLM. These observations reinforce the need for developing approaches that enhance the robustness and effectiveness of integration testing for REST APIs.

1.4 Objectives and Research Questions

The main objective of this work is to introduce RestTSLLM, an approach for the automatic generation of integration tests for REST APIs based on their OpenAPI Specification and using LLMs. To simplify the understanding by LLMs by decomposing the problem (KHOT et al., 2023; SADOWSKI; CHUDZIAK, 2026), we suggest the use of an intermediate language, capable of structuring business specifications into structured test scenarios that can address all flows, validating both happy paths (expected path to success) and edge cases, including validation failures. For this purpose, we propose the use of Test Specification Language (TSL) (OSTRAND; BALCER, 1988), which serves as a bridge between business requirements and automated testing by providing a high-level, declarative format for specifying test cases. Instead of writing low-level test scripts, users define inputs, expected behaviors, and outcomes in a structured, human-readable format. By abstracting the test logic from its implementation, TSL enables the automatic generation of test scripts for different platforms, making it particularly effective for validating APIs and complex systems through reusable and maintainable specifications. Additionally, this study aims to evaluate the effectiveness of LLMs in understanding the context of these test scenarios and generating the corresponding set of integration tests. Thus, this research seeks to answer the following Research Questions (RQs):

[1] OpenAPI Specification (OAS) is a specification for describing REST APIs in a standardized, and machine-readable format.

RQ1: Are LLMs effective in generating integration tests that reflect the intended business logic and context?
It is addressed through a qualitative analysis of all generated test cases—examining whether they are compilable, contextually coherent, and aligned with the OpenAPI Specification—and through objective metrics (success rate, coverage, and mutation score) collected by executing the tests against running REST API instances.

RQ2: Which LLM is the most effective in generating integration tests?
Effectiveness is operationalized as the combined performance across success rate, coverage, and mutation score, aggregated into a single calculated score for each model. The models are ranked by score, and the one with the highest value across all six evaluated projects is identified as the most effective.

RQ3: What is the level of acceptance of the RestTSLLM approach among practitioners involved in the REST API development lifecycle?
It is addressed through an acceptance survey applying a Likert-scale instrument to practitioners involved in the REST API development lifecycle, measuring perceived utility, trust in the generated tests, intention to use, and industrial applicability, complemented by an inductive open-coding analysis of qualitative responses.

1.5 Method Overview

To answer these Research Questions, this Master's dissertation proposes a new approach, conducts a comparative experimental evaluation, and conducts a survey with practitioners to validate the applicability of the proposed approach.

The proposed approach, RestTSLLM, adopts prompt engineering with the *few-shot* and *decomposed prompting* techniques (KHOT et al., 2023; SADOWSKI; CHUDZIAK, 2026), teaching the LLM, part by part, how to produce appropriate responses to the prompts submitted to the model. Our approach includes an intermediate step for generating integration tests. First, we convert an OpenAPI Specification into test cases using the TSL. Then, we convert the TSL test cases into functional integration tests using xUnit (.NET) (MICROSOFT, 2025). In presenting examples to the model, we demonstrate to the LLM how to follow these steps.

The experiment also aims to execute the prompts across multiple projects and models previously selected based on criteria established in this study. After processing all prompts

and generating the integration tests, we execute the tests produced by the models and collect and evaluate performance metrics, allowing us to answer the RQ1 and RQ2.

Finally, we conducted a survey with practitioners working in the REST API development lifecycle to present the approach and explain how it works, map the participants' profiles, and assess their intention to use the approach, aiming to answer RQ3.

1.6 Main Results

As a result of this experiment applied to six open-source projects, the models Claude 3.5 Sonnet (Anthropic), Deepseek R1 (Deepseek), Qwen 2.5 32b (Alibaba), and Sabiá 3 (Maritaca) achieved the best results, and met our expectations, demonstrating strong potential as promising tools for this purpose. These tools enabled us to generate more effective integration tests than the other evaluated LLMs. Claude 3.5 Sonnet outperformed all other models across every metric, emerging as the most suitable model for this task. The models Mistral Large (Mistral), Gemini 1.5 Pro (Google), GPT 4o (OpenAI), and LLaMA 3.2 90b (Meta) performed lowest but were still considered satisfactory.

Beyond the technical evaluation, the acceptance survey conducted with 96 practitioners involved in the REST API development lifecycle (RQ3) showed a high level of acceptance of the approach, with an overall agreement of 4.28 on a 1–5 scale and agreement of at least 74% on every evaluated dimension. Practitioners particularly valued the usefulness of the approach and its intermediate TSL representation, and expressed a strong intention to adopt it; their main reservation was trust in the generated tests and the consequent need for human review. A complementary qualitative analysis of the open-ended responses corroborated these findings and yielded a set of concrete, practitioner-driven directions for evolving the approach.

1.7 Contributions

This work presents as its main contributions the proposed approach, RestTSLLM, for integration test generation using TSL and LLMs; the performance comparison of different LLMs in generating integration tests according to our method, identifying those with the best results; and the development of an automated script capable of integrating multiple LLMs, which also allows easy connection to additional models for future research and reuse of the code, enabling a wide range of experiments in a faster and more adaptable

manner across different usage contexts.

Additionally, this dissertation contributes an acceptance survey conducted with practitioners involved in the REST API development lifecycle, providing empirical evidence on the perceived utility, adoption intention, and preferred integration modalities for the RestTSLLM approach.

Part of the work presented in this dissertation — including the RestTSLLM approach and the experimental evaluation across six open-source projects — was published as a full paper at the 39th Brazilian Symposium on Software Engineering (SBES 2025) (BARRADAS; PAES; NEVES, 2025).

1.8 Availability of Data and Code

The data and code supporting the conclusions of this study are publicly available on GitHub (BARRADAS, 2025).

1.9 Benefits and Relevance of the Approach

The RestTSLLM approach aims to support researchers, developers, and companies in improving REST API testing. Automating test generation can reduce time, cost and effort while enhancing fault detection. The flexibility of LLMs also facilitates faster adaptation to evolving requirements. Finally, this study fills a gap in the literature by evaluating the use and performance of LLMs in REST API testing.

1.10 Dissertation Structure

This chapter (1) establishes the main research objectives, presenting a brief introduction to the context of software testing, its challenges, integration testing with REST APIs, studies with LLMs, their open gaps, a summary of the methodology, contributions, and results obtained.

The remainder of this dissertation is organized as follows: Chapter 2 provides more in-depth context on integration testing and REST API. Chapter 3 reviews related work. Chapter 4 presents more details about our approach, RestTSLLM, and how it can be replicated. Chapter 5 describes the experimental evaluation applied, including details

of the selected technologies and tools, the prompts utilized, and the evaluation and comparison metrics used in this study. Chapter 6 explains details of survey conducted with practitioners in the field about RestTSLLM approach acceptance. Chapter 7 delves into analyzing the results obtained in the experiment, analyzes the survey results, and discusses threats to the validity and limitations of the study regarding its applicability. Finally, Chapter 8 presents the conclusions, summarizing the main contributions and outlining possible future work.

2 Background

This chapter introduces the foundational concepts required to understand the context and contributions of this dissertation. Section 2.1 provides an overview of software testing, its phases, and core techniques. Section 2.2 deepens the discussion on integration testing and how it applies to REST APIs. Section 2.3 presents the Test Specification Language (TSL) and the Category-Partition method on which it is based. Section 2.4 introduces Artificial Intelligence (AI), Large Language Models (LLMs), and their role in software engineering. Finally, Section 2.5 describes prompt engineering and the techniques adopted in this work.

2.1 Software Testing

Software testing is the process of evaluating a software system—or one of its components—by executing it under specified conditions and comparing the observed behavior with the expected behavior (TUTEJA; DUBEY, et al., 2012; DELAMARO; JINO; MALDONADO, 2013). Software testing is a verification and validation activity whose primary goal is to reveal errors in a system, exercising it under controlled conditions and comparing its actual behavior against the expected results (DELAMARO; JINO; MALDONADO, 2013). These goals are pursued across different stages of development and through different strategies, each targeting a distinct aspect of the system.

Testing plays a central role in ensuring the quality and reliability of systems, and its importance has grown proportionally to the increasing complexity of modern software (TUTEJA; DUBEY, et al., 2012; JINDAL, 2016; ANAND; UDDIN, 2019). Exhaustively testing all possible inputs and scenarios is computationally infeasible for most real-world systems, which makes the selection of representative and high-impact test cases a critical engineering decision (TUTEJA; DUBEY, et al., 2012; ORSO; ROTHERMEL, 2014). Inadequate testing can result in undiscovered failures, high maintenance costs, and, in critical systems, severe consequences for users and organizations (JINDAL, 2016).

2.1.1 Phases of Testing

Testing activities are typically organized into phases that reflect the granularity and scope of the components being verified. These phases progress from narrowly focused evaluations of individual units of code to broad assessments of complete systems. The most widely recognized phases are described below.

Unit Testing evaluates individual functions, methods, or classes in isolation from the rest of the system (SOMMERVILLE, 2016; SOTIRIADIS et al., 2017). In unit testing, dependencies such as databases, network services, or other modules are typically replaced by test doubles (mocks, stubs, or fakes) to ensure that a test fails only due to the behavior of the specific unit under examination. Although unit tests can be written and executed quickly, they do not verify how components interact and therefore cannot detect failures that arise at component boundaries.

Integration Testing evaluates the interaction between two or more components that have been combined into a subsystem (DELAMARO; JINO; MALDONADO, 2013; NEWMAN, 2015). Unlike unit tests, integration tests exercise real collaboration between components—including communication protocols, data formats, and shared state—and are therefore more sensitive to interface mismatches and emergent behaviors. This phase is discussed in greater depth in Section 2.2.

System Testing evaluates the entire integrated system against its specified requirements (DELAMARO; JINO; MALDONADO, 2013). Tests at this phase exercise end-to-end workflows and may include functional, performance, security, and usability tests. System tests reflect the perspective of a user or external client interacting with the fully assembled product.

Acceptance Testing is performed to determine whether the system satisfies the acceptance criteria agreed upon with the stakeholder or customer (DELAMARO; JINO; MALDONADO, 2013). Acceptance tests may be written and executed by the development team (as in Behavior-Driven Development (BDD)), or by the customer directly (as in user acceptance testing), and they serve as the final quality gate before the system is released.

2.1.2 Test Design Techniques

A test oracle—the mechanism used to determine whether a test passed or failed—is a prerequisite for any test. Once oracles are defined, test design techniques guide how inputs

and scenarios are selected to maximize the likelihood of exposing defects within a practical number of test cases.

Black-box testing (also called functional testing or specification-based testing) derives test cases exclusively from the specification of the system, without knowledge of its internal implementation (DELAMARO; JINO; MALDONADO, 2013). Black-box techniques include:

- *Equivalence Partitioning*: divides the input domain into classes of values that are expected to be treated identically by the system. One representative value per class is sufficient to detect a given type of fault (DELAMARO; JINO; MALDONADO, 2013).
- *Boundary Value Analysis*: selects test inputs at the edges of equivalence classes—the minimum, maximum, and values just outside the valid range—where faults are statistically more common (DELAMARO; JINO; MALDONADO, 2013; SOMMERVILLE, 2016).
- *Decision Table Testing*: constructs a table of all relevant combinations of conditions and their corresponding actions, ensuring that every business rule is exercised at least once.

White-box testing (also called structural testing) derives test cases from the internal structure of the code, aiming to ensure that specific structural elements—such as statements, branches, or paths—are exercised. Coverage metrics such as statement coverage, branch coverage, and condition coverage are used to measure and guide white-box test design.

Both families of techniques are complementary and often used together in practice: black-box techniques ensure that the system satisfies its specification, while white-box testing techniques ensure that the implementation is adequately exercised.

2.1.3 REST APIs and the HTTP Protocol

Representational State Transfer (REST) is an architectural style for distributed hypermedia systems, characterized by a stateless client-server communication model in which resources are identified by URLs and manipulated through a uniform interface (FIELDING, 2000; LERCHER, 2024).

Following the model proposed by Fielding (2000), two foundational concepts underpin this style:

- A **resource** is any named piece of information that can be identified, addressed, and manipulated—a user account, a product, a collection of orders, or any other concept meaningful to the application.
- A **representation** is a snapshot of the resource's current or intended state, transferred between client and server in a negotiated format. REST APIs typically use JavaScript Object Notation (JSON) as the representation format, although other encodings (e.g., XML) are also valid.

The resource and its representations are distinct: the same resource (e.g., `GET /books/42`) may have multiple representations, and the client manipulates the resource's state solely by exchanging representations through the uniform interface—never by accessing the resource directly.

In practice, REST APIs are built on top of the Hypertext Transfer Protocol (HTTP) protocol and use its standard methods to convey the intent of each request:

- `GET` — retrieves a resource or a collection of resources.
- `POST` — creates a new resource.
- `PUT` / `PATCH` — replaces or partially updates an existing resource.
- `DELETE` — removes a resource.

Responses include an HTTP status code that communicates the outcome of the request (e.g., `200 OK`, `201 Created`, `400 Bad Request`, `401 Unauthorized`, `404 Not Found`), and a body—typically in JavaScript Object Notation (JSON)—that carries the returned data or error details.

The behavior of a REST API is often documented using the OpenAPI standard (formerly Swagger), a machine-readable specification format that describes available endpoints, parameters, request bodies, response schemas, and authentication requirements. The OpenAPI Specification (OAS) (JSON) is the primary artifact used as input in this dissertation's approach.

An example of OAS is shown in Figure 1, which displays the official demonstration project used to teach, test, and document the standard, both the JSON (on the left) and the rendering (on the right) used to make the documentation available to people.

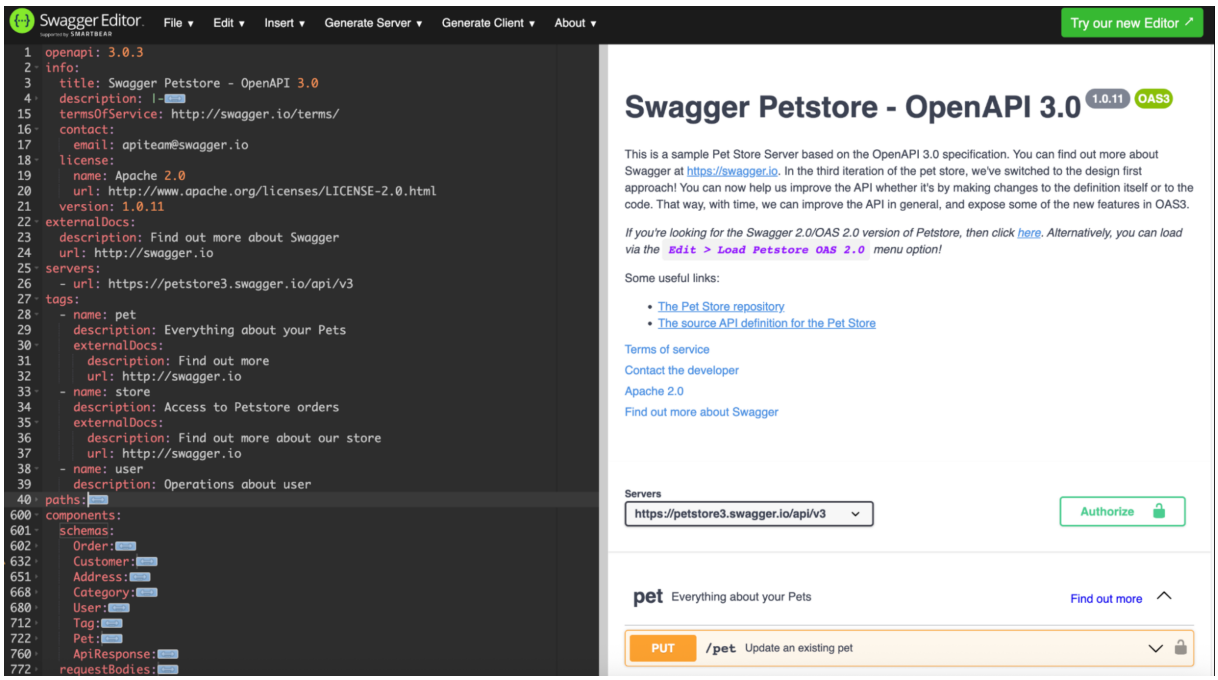

Figure 1: Example of an OpenAPI Specification (OAS) from Petstore. The JSON is on the left and a rendered view is on the right

2.2 Integration Testing for REST APIs

Integration tests focus on the interactions between components and verify that they cooperate correctly when combined (DELAMARO; JINO; MALDONADO, 2013; NEWMAN, 2015). Unlike unit tests, which replace dependencies with controlled substitutes, integration tests exercise real collaborations—including communication protocols, serialization formats, database queries, and authentication flows. This makes them more representative of production behavior but also more complex to design and maintain (GOLMOHAMMADI; ZHANG; ARCURI, 2023).

Newman (2015) emphasizes that in distributed architectures—where independent services communicate over a network—failing to properly validate these interactions can lead to critical failures in production, even when all individual components pass their unit tests. Microservices often interact via REST APIs, and much of their behavior emerges from these interactions; consequently, unit-level testing alone cannot capture the interconnected, system-level behavior and its dependencies, nor the faults arising from inter-service interactions (CAO; PANICHELLA; VERWER, 2025). Validating REST API behavior in realistic scenarios thus depends on integration tests that exercise these interactions (REILE et al., 2022). These dependencies and architectural complexities can be seen in Figure 2, which demonstrates a client calling a REST API that, in turn,

relies on another REST API, an authentication service, and connections to a database, cache, and message broker—a simple architectural design with realistic complexities and challenges for integration testing.

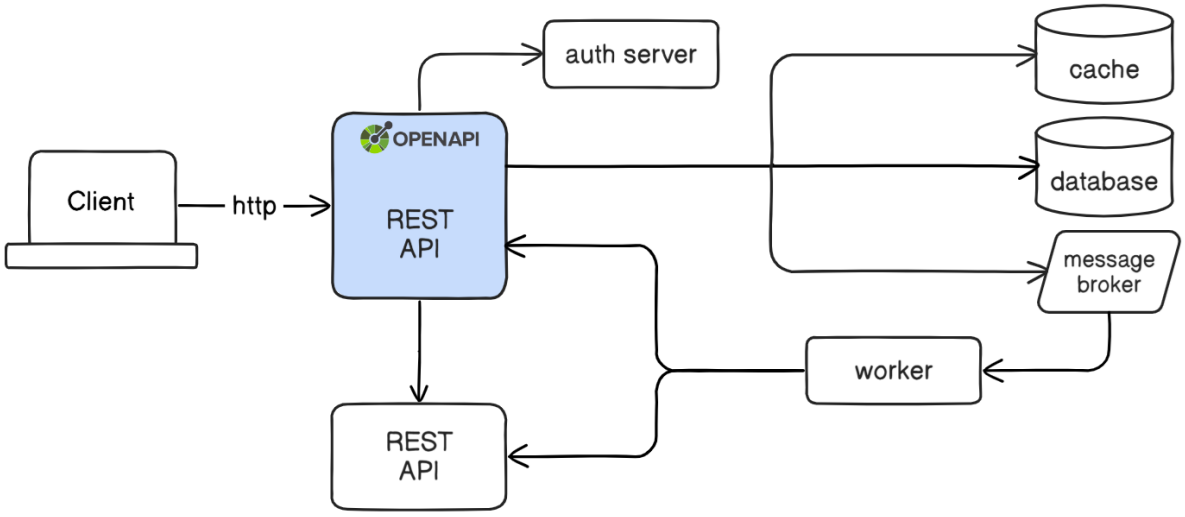

Figure 2: Example of REST API architecture with dependencies

2.2.1 Characteristics of REST API Integration Tests

An integration test for a REST API exercises one or more *endpoints* — the individual operations the API exposes, each identified by an HTTP method and a URL path (e.g., `POST /users`) — through real HTTP requests directed at a running server instance that has access to an actual (or test-specific) database and any required external services. The test verifies not just the HTTP status code but also the structure and semantics of the response body, side effects in persistent storage, authentication and authorization behavior, and inter-endpoint dependencies (e.g., a `DELETE` test that first creates the resource via `POST`).

Well-designed REST API integration tests typically follow the Arrange, Act, Assert (AAA) pattern (WEI, C. et al., 2025):

1. **Arrange**: set up the preconditions—create necessary entities, authenticate, and configure the environment.

2. **Act**: execute the function or functionality (for a REST API, the HTTP request) under test.

3. **Assert**: verify that the response status code, headers, and body match the expected behavior; optionally, verify that the database state was updated correctly.

Generating comprehensive suites of tests for real-world APIs requires defining many scenarios—valid inputs, boundary values, missing fields, wrong authentication, etc.—and implementing the corresponding setup, assertions, and teardown for each. This process is time-consuming and error-prone when done manually, which motivates the automation approach proposed in this dissertation.

2.2.2 Challenges in REST API Integration Testing

Several factors make integration testing for REST APIs particularly challenging (GOLMOHAMMADI; ZHANG; ARCURI, 2023):

- **Input space complexity**: REST APIs accept structured payloads with multiple fields, each subject to its own type, format, and range constraints. Exhaustively testing all combinations is infeasible.
- **Stateful dependencies**: many test scenarios require an initial state that must be established through prior requests (e.g., creating an entity before deleting it), introducing ordering constraints between tests.
- **Authentication**: REST APIs often protect endpoints with token-based or session-based authentication, requiring tests to handle token acquisition as part of the *Arrange* phase.
- **Oracle definition**: determining the correct expected response for a given input requires a thorough understanding of the business rules and system contracts, which may not be fully captured in the OpenAPI specification.
- **Input data generation**: generating semantically valid inputs (e.g., realistic emails, correctly formatted dates, or values within business-specific ranges) that go beyond syntactic validity defined in the schema is a non-trivial task.

2.3 Test Specification Language (TSL)

Test Specification Language (TSL) is a formalism for describing test cases at an abstract level, independently of any particular programming language or testing framework.

It provides a structured, human-readable format for capturing the *what* of a test—its preconditions, inputs, and expected outcomes—without prescribing the *how* of its implementation (OSTRAND; BALCER, 1988).

2.3.1 The Category-Partition Method

TSL originates from the *Category-Partition method*, introduced by Ostrand and Balcer (1988). This method is a black-box test design technique that derives test cases systematically from a functional specification by following four steps:

1. **Identify categories**: analyze the specification of the unit under test and identify its parameters and environmental conditions (the *categories*).
2. **Partition each category into choices**: for each category, enumerate the distinct input classes that are expected to trigger different behaviors (the *choices*).
3. **Determine constraints**: define constraints between choices that restrict invalid or redundant combinations (e.g., "if the user is unauthenticated, skip scenarios that require a valid token").
4. **Generate test frames**: produce a set of test frames by selecting one choice per category according to the constraints, ensuring that each meaningful combination is covered.

This method enables testers to derive a systematic, manageable set of test cases that covers all relevant input classes and their combinations, without requiring exhaustive enumeration.

2.3.2 TSL Structure and Format

TSL test cases are expressed in a structured format such as *yaml*. Each test case specifies the function (method, endpoint etc) to be exercised, the preconditions that must hold before the execution, the input parameters, and the expected output.

The declarative nature of TSL makes it easy for a reader—human or automated tool—to understand the intent of each test case without reference to a specific implementation technology. Because TSL specifications describe *what* should be tested rather than *how* the test should be executed, they can be translated into executable tests for different languages or frameworks, making the notation inherently technology-agnostic.

2.3.3 Role of TSL in the RestTSLLM Approach

In RestTSLLM, TSL serves as a bridge between two phases of generation: (1) deriving business-level test scenarios from an OpenAPI specification, and (2) translating those scenarios into runnable integration test code. By decomposing the problem into these two steps, the approach follows the decomposed prompting strategy (KHOT et al., 2023; SADOWSKI; CHUDZIAK, 2026), which reduces the cognitive complexity of each step and improves the quality of the final output. The rationale and implementation of this design are detailed in Chapter 4.

2.4 Generative AI and Large Language Models

Artificial Intelligence (AI) is a broad field of computer science that studies the design of systems capable of performing tasks that typically require human intelligence, such as perceiving, reasoning, learning, and generating natural language (FAN et al., 2023). Within AI, *Machine Learning* (ML) is the sub-field that develops algorithms capable of improving their performance on a task through experience, without being explicitly programmed for every case. *Natural Language Processing (NLP)* is an ML sub-discipline specifically concerned with the understanding and generation of human language.

2.4.1 Large Language Models

Large Language Models (LLMs) are a class of neural network models trained on massive corpora of text with the objective of predicting the probability distribution of the next token given a preceding sequence. Contemporary LLMs are built on the *transformer* architecture, which relies on an attention mechanism to capture long-range dependencies in text and process inputs in parallel (HOU et al., 2024). During training, the model learns latent representations of syntax, semantics, factual knowledge, and reasoning patterns from a diverse collection of documents, books, code repositories, and web pages.

Through this training process, LLMs acquire capabilities that extend beyond simple text completion. They can follow instructions, answer questions, write and debug code, summarize documents, translate between languages, and perform logical reasoning—all from a natural-language description of the task, without requiring any modification of the model's parameters (HAGOS; BATTLE; RAWAT, 2024; HOU et al., 2024). This zero-shot or few-shot adaptability distinguishes LLMs from earlier ML models, which

typically required dedicated training data and task-specific architectures.

Well-known LLMs include GPT 4o, LLaMA 3.2 90b, Claude 3.5 Sonnet, Gemini 1.5 Pro, Deepseek R1, Mistral Large, Qwen 2.5 32b, and Sabiá 3, among others. These models vary in size, training methodology, licensing terms, and the types of tasks they handle best. In this dissertation, a comparative evaluation of the performance of these models on integration test generation is conducted, as described in Chapter 5.

2.4.2 Limitations of LLMs

Despite their capabilities, LLMs present important limitations that must be considered when applying them to engineering tasks (FAN et al., 2023; BOUKHLIF; KHARMOUM; HANINE, 2024):

- **Hallucination**: LLMs can produce plausible-sounding but incorrect or inconsistent outputs, which in a testing context can result in test cases that do not correctly reflect the specification.
- **Output length limits**: models have a maximum number of output tokens per response, which can lead to truncated outputs when generating large test suites.
- **Sensitivity to prompt phrasing**: small changes in the wording of a prompt can lead to substantially different outputs, making reproducibility and consistency a concern.
- **Stochasticity**: the probabilistic nature of text generation means that repeated executions of the same prompt may produce different outputs, introducing variability in quality.
- **Context window**: the model can only attend to a limited number of tokens at once; very large specifications may need to be segmented before being submitted as input.

These limitations motivate the use of structured prompt engineering strategies—described in the following section—to guide LLMs toward more reliable and accurate outputs.

2.5 Prompt Engineering

A prompt is the textual input provided to an LLM to elicit a specific response. The prompt encodes the task description, relevant context, constraints, and any examples that help the model produce the desired output (MU et al., 2025; ZHANG, L. et al., 2024). Prompt engineering is the discipline of designing and structuring prompts to maximize the quality, accuracy, and reliability of the model's response. It does not involve modifying the model's weights; rather, it operates entirely at the interface between the user and the model.

2.5.1 Prompt Types

Modern LLM APIs distinguish between two main prompt types, which serve complementary roles (NEBULY, 2024):

- **System prompt**: provided before the conversation begins, it configures the model's global behavior—defining a persona, restricting the scope of responses, setting the output format, and establishing constraints that apply to all subsequent interactions. For example, a system prompt may instruct the model to act as a software testing expert and to produce output exclusively in a specified structured format.
- **User prompt**: the task-specific instruction provided at each conversational turn, containing the input data (e.g., an OpenAPI specification), the expected output format, and any additional constraints relevant to that specific step.

2.5.2 Prompting Techniques

Several prompting techniques have been proposed to improve the quality of LLM outputs (OUÉDRAOGO et al., 2024):

Zero-shot prompting provides only the task description, relying entirely on the model's pre-trained knowledge to produce the output. It requires no additional examples and is the simplest form of prompting. However, for complex or domain-specific tasks, zero-shot performance may be insufficient (OUÉDRAOGO et al., 2024).

Few-shot prompting supplements the task description with a small number of labeled input-output examples ("shots"). These examples illustrate the expected structure and conventions of the output, effectively teaching the model through demonstration rather

than explicit instruction. Few-shot prompting consistently improves performance on tasks that require adherence to specific formats or domain conventions, such as generating tests in a particular framework (OUÉDRAOGO et al., 2024).

Chain-of-thoughts prompting encourages the model to produce a step-by-step reasoning process before arriving at the final answer. By externalizing the intermediate reasoning steps, this technique improves correctness on multi-step or arithmetically complex problems (WEI, J. et al., 2022).

Tree-of-thoughts prompting extends the chain-of-thoughts concept to explore multiple reasoning branches simultaneously, scoring each branch and selecting the most promising path. It is particularly useful for tasks with combinatorial solution spaces (YAO et al., 2023).

Decomposed prompting (KHOT et al., 2023; SADOWSKI; CHUDZIAK, 2026) decomposes a complex task into a sequence of simpler subtasks, each addressed by a dedicated prompt. This approach reduces the cognitive load per prompt, allows the model to specialize on a narrower objective at each step, and improves the reproducibility and interpretability of the output.

3 Related Work

This chapter reviews relevant studies that explore the use of LLMs in software engineering, with a particular focus on test automation and REST API testing, and provides a summary of the identified research gaps that motivate this study. The chapter is organized from the general to the specific: Section 3.1 surveys the broad use of LLMs in software engineering; Section 3.2 narrows the scope to test data generation; Section 3.3 focuses specifically on REST API testing with LLMs; Section 3.4 covers traditional approaches that do not rely on LLMs, establishing the baseline against which LLM-based approaches are compared; and Section 3.5 synthesizes the identified research gaps that motivate the present work.

3.1 Use of LLMs in Software Engineering

There is a growing interest in using LLMs across various areas of software engineering, including requirement engineering, code implementation, testing, maintenance, and deployment (HOU et al., 2024; FAN et al., 2023). Fan et al. (2023) provide a comprehensive overview of this trend, highlighting the diverse applications of LLM tools in tasks such as requirements, coding, bug fixing, refactoring, performance improvement, design, documentation, and analysis. Moreover, the study points out key technical challenges that come with these advances, particularly the need for reliable methods to detect and correct incorrect solutions.

The study by Belzner, Gabor, and Wirsing (2023) provides an insightful overview of the benefits and challenges of using LLMs in software engineering, focusing on specific stages of the development cycle such as requirements engineering, system design, code generation, and testing. The results indicate that LLMs–GPT 3.5 and Bard (now Gemini)–can effectively assist in creating helpful software engineering artifacts, particularly during simpler phases of development. However, they present limitations in terms of scalability and accuracy, especially in more complex or ill-defined tasks such as integration testing. Although the study analyzes various development lifecycle phases, it provides limited

attention to integration testing and complex testing scenarios.

Zheng et al. (2025) conducted a systematic review of 123 LLM-for-SE papers published from 2022 onward, categorizing them across seven software engineering tasks: code generation, code summarization, code translation, vulnerability detection, code evaluation, code management, and QA interaction. Their analysis reveals that LLMs perform with high confidence in syntax-oriented tasks such as code summarization and program repair, but tend to exhibit lower confidence in semantically complex tasks such as code generation and vulnerability detection. Regarding test case generation, the study reports a generally positive outlook among researchers, while acknowledging that further empirical validation is needed across diverse contexts. This observation reinforces the relevance of comparative empirical studies that evaluate LLM effectiveness in more specialized testing domains.

A broader perspective is offered by Quanjun Zhang et al. (2026), whose comprehensive survey synthesizes 988 LLM-based SE studies covering five phases of the software lifecycle – requirements and design, development, testing, maintenance, and management – and catalogs 62 code-oriented LLMs across three architectural families. The authors document a sharp increase in publications, from 13 papers in 2020 to 489 in 2024, reflecting the field's rapid growth. The survey also identifies open challenges such as model versioning, benchmarking standardization, and the security and reliability of LLM-generated artifacts. These challenges remain particularly relevant for empirical studies that compare multiple LLMs, since results obtained with one model version may not necessarily generalize to future releases.

3.2 LLMs for Test Data and Test Generation

In comparative performance studies, Mendoza et al. (2024) conducted a comparative analysis of different LLM tools, including GPT 3.5, GPT 4, Copilot, and Gemini, to assess their ability to generate input data from Behavior-Driven Development (BDD) scenarios. The study highlighted that GPT 4 and Gemini performed better in generating test data consistent with BDD specifications, outperforming GPT 3.5 and Copilot. Similarly to the present work, this study adopts a comparative multi-model perspective; however, it focuses on input data generation from behavioral specifications, whereas our study investigates the generation of complete executable integration tests from OpenAPI Specification documents.

Extending the use of LLMs for test input generation, Tu et al. (2025) proposed a hybrid approach that combines concolic execution with LLMs to generate highly structured inputs

for formats such as JSON, XML, SQL, and JavaScript. Rather than relying solely on prompting, the approach leverages program analysis techniques to guide input generation and improve path exploration, demonstrating improvements in coverage and vulnerability detection. While both approaches deal with structured data formats and test input generation, they address different objectives. Cottontail focuses on generating valid structured inputs through the combination of symbolic reasoning and LLMs, whereas this work focuses on generating reusable integration test artifacts for REST APIs using only prompt engineering and information available in the OpenAPI Specification.

The work by Ouédraogo et al. (2024) presents a comprehensive analysis of LLMs in the generation of unit tests, exploring the effectiveness of different models such as GPT 3.5, GPT 4, Mixtral 7B, and Mixtral 8x7B, along with advanced prompt engineering techniques, including *zero-shot*, *few-shot*, *chain-of-thoughts*, and *tree-of-thoughts*. The results indicated that GPT 3.5 achieved the best performance. Furthermore, the study revealed that although LLMs show significant potential, there are still limitations related to the quality of the generated tests and the presence of test smells that may compromise long-term maintainability. Even though the study focuses on unit testing—which is inherently simpler and requires less contextual information than integration testing—it highlights the need to evolve test generation methods and refine prompting strategies to improve test quality, coverage, and maintainability. These challenges become even more relevant in integration testing scenarios, where interactions among multiple system components must also be considered.

Complementing these findings, Chang and Shirazi (2025) proposed a systematic benchmark for assessing the capability of LLMs to generate test cases under different software structures and control-flow patterns. Their evaluation showed that state-of-the-art models can effectively generate tests for common scenarios but still struggle with programs requiring more sophisticated logical reasoning and arithmetic operations. The study also highlights the need for evaluation criteria that go beyond traditional coverage metrics when assessing LLM-generated tests. This observation aligns with the evaluation strategy adopted in this work, which considers multiple dimensions, including success rate, coverage, mutation score, and a weighted composite score.

Recent studies have moved beyond prompt-only approaches. Gu, Nashid, and Mesbah (2025) proposed an iterative framework that combines LLMs, static analysis, dynamic analysis, and coverage feedback to guide automated test generation. Their results demonstrated substantial gains in line and branch coverage when compared to previous

approaches, indicating that hybrid techniques represent a promising direction for improving the effectiveness and reliability of LLM-based test generation. Unlike PANTA, which relies on iterative execution, program analysis, and feedback loops, our work investigates how far test generation can be pushed using only prompt engineering and information available in the OpenAPI Specification. This distinction allows us to evaluate the capabilities of general-purpose LLMs in a simpler, more accessible, and reproducible setting.

Liu et al. (2025) investigated whether LLMs can generate regression tests directly from software commits and patches. Their approach, called Cleverest, employs zero-shot prompting to produce structured test inputs capable of reproducing bugs or validating fixes. The study demonstrated that LLMs can successfully generate meaningful regression tests without requiring manually crafted seed inputs, achieving competitive results when compared to traditional testing techniques. While Cleverest focuses on software evolution and regression testing, our work addresses a different challenge: generating reusable integration tests directly from OpenAPI Specifications. Nevertheless, both studies explore the potential of general-purpose LLMs to automate testing activities without requiring model fine-tuning or specialized training procedures.

Overall, recent research demonstrates that LLMs are increasingly capable of generating both test inputs and executable test cases. However, most studies focus on unit testing, structured input generation, regression testing, or hybrid analysis techniques. Comparatively less attention has been given to the automated generation of reusable integration tests, particularly in the context of REST APIs. Furthermore, many recent advances rely on static analysis, dynamic analysis, execution-feedback loops, or specialized models, leaving open the question of how effective modern general-purpose LLMs can be when guided exclusively through prompt engineering and information extracted from OpenAPI Specifications.

3.3 LLMs for REST API Testing

Recent research has increasingly explored the use of LLMs to improve automated testing of REST APIs, particularly by leveraging the natural-language descriptions available in OpenAPI Specifications (KIM; STENNETT; SHAH, et al., 2024; KIM; SINHA; ORSO, 2025). Traditional black-box tools typically use the structured elements of the specification, such as endpoints, parameters, schemas, and status codes, but often struggle to interpret semantic constraints, business rules, and examples described in natural language

(GOLMOHAMMADI; ZHANG; ARCURI, 2023; KIM; STENNETT; SHAH, et al., 2024). This limitation has motivated a new line of work in which LLMs are used to extract constraints, generate realistic inputs, identify dependencies, guide test execution, and produce executable test artifacts.

Kim, Stennett, Shah, et al. (2024) proposed RESTGPT, one of the first approaches to explicitly leverage LLMs to improve REST API testing from OpenAPI Specifications. RESTGPT parses an OpenAPI Specification and uses an LLM to extract machine-interpretable rules, constraints, and example parameter values from the human-readable descriptions in the specification. The resulting enriched specification can then be used by traditional automated testing tools. The study reports substantial improvements over NLP-based and knowledge-base-based approaches, such as NLP2REST and ARTE, particularly in rule extraction and generation of syntactically and semantically valid input values. However, RESTGPT does not aim to generate complete executable test suites. Instead, it improves the raw input used by downstream testing tools. In contrast, the present work uses the OpenAPI Specification as the primary source for generating reusable and executable integration tests, rather than only enriching the specification for other tools.

Expanding this line of research, Kim, Sinha, and Orso (2025) proposed LlamaRestTest, a black-box testing REST API testing approach based on fine-tuned and quantized LLaMA 3 models. The approach employs two specialized models: one for detecting inter-parameter dependencies and another for generating realistic input values. Unlike RESTGPT, which statically enriches the OpenAPI Specification, LlamaRestTest dynamically analyzes server responses during execution and uses this feedback to refine test generation. The results show that small specialized models can outperform larger general-purpose models and traditional tools in coverage and fault detection. However, this approach requires model training, quantization, and dynamic interaction with the System Under Test (SUT). The present work investigates a different trade-off: instead of optimizing runtime exploration through specialized models and server feedback, it evaluates how far general-purpose LLMs can be pushed through prompt engineering and a structured TSL-based intermediate representation, while generating reusable test artifacts that can be versioned and integrated into existing development workflows.

A complementary dependency-aware direction is explored by Le et al. (2024), who proposed KAT, an automated REST API testing approach using GPT and advanced prompting techniques. KAT constructs an Operation Dependency Graph from the

OpenAPI Specification and uses this graph to generate operation sequences, test scripts, test data, and constraint validation scripts. The approach explicitly addresses dependencies among operations, dependencies among parameters, and dependencies between operations and parameters. This is particularly relevant for stateful REST APIs, where a valid test sequence may require creating or retrieving resources before invoking dependent operations. While KAT and the present work both use OpenAPI Specifications and general-purpose LLMs without fine-tuning, they differ in their intermediate representation and execution model. KAT relies on explicit dependency graphs to guide operation sequencing, whereas this work uses a TSL-based intermediate representation to structure test scenarios, preconditions, inputs, and expected outcomes before translating them into executable tests.

Building on the broader trend of dependency-aware and feedback-driven REST API testing, Kim, Stennett, Sinha, et al. (2025) introduced AutoRestTest, a multi-agent approach that combines LLMs, a Semantic Property Dependency Graph, and Multi-Agent Reinforcement Learning to optimize REST API exploration. AutoRestTest decomposes the testing process into specialized agents responsible for operation selection, dependency handling, parameter selection, and value generation. The approach uses semantic similarity to reduce the search space for operation dependencies and reinforcement learning to adapt exploration strategies based on observed responses. Its evaluation on real-world REST services shows improvements in coverage and fault detection when compared to state-of-the-art black-box testing tools. AutoRestTest represents a highly sophisticated feedback-driven strategy for maximizing exploration effectiveness at runtime. In contrast, the present work targets a different objective: generating reusable, versionable integration tests from the OpenAPI Specification alone, without requiring a running service during generation — producing persistent artifacts that developers can inspect, maintain, or integrate into CI/CD pipelines, rather than ephemeral exploration sequences. The TSL-based intermediate representation adopted in this work makes the generated test logic explicit and human-readable — a design choice that prioritizes reproducibility and workflow integration over runtime exploration depth.

Another recent contribution close to the objective of this dissertation is APITestGenie, proposed by Pereira, Lima, and Faria (2026). APITestGenie generates executable API integration tests from business requirements written in natural language and OpenAPI Specifications, using LLMs, Retrieval-Augmented Generation (RAG), and prompt engineering. The tool was evaluated on real-world APIs, including industrial services, and the results indicate that it can generate syntactically and semantically valid test scripts for a

high proportion of the business requirements under analysis. APITestGenie is particularly relevant because it targets executable integration tests and uses high-level requirements to guide test generation, which brings the generated tests closer to business intent. However, it assumes the availability of explicit business requirements as an additional input artifact, going directly from requirements and OpenAPI Specification to executable tests in a single generation step. The present work follows a different design: rather than requiring explicit requirements, it introduces a TSL-based intermediate representation that first extracts structured test scenarios directly from the OpenAPI Specification before producing executable code. This separation between scenario reasoning and code generation — the core of the decomposed prompting strategy — reduces the cognitive load on the LLM at each step and eliminates the dependency on separately maintained requirements documentation, which may be incomplete, outdated, or unavailable in practice.

Kogler, Ehrhart, et al. (2025) proposed RESTifAI, an LLM-based workflow for generating reusable and CI/CD-ready REST API tests from OpenAPI Specifications. RESTifAI focuses on constructing valid happy-path scenarios and deriving negative test cases to validate both expected functionality, represented by 2xx responses, and robustness against invalid inputs or business-rule violations, represented by 4xx responses. This makes RESTifAI one of the closest works to the present study, since both approaches focus on reusable test artifacts and consider both successful and client-error scenarios. Nevertheless, the approaches differ in how test scenarios are derived. RESTifAI relies on workflow execution and server feedback to guide the construction of successful paths, from which negative cases are subsequently derived. The present work takes a different approach: a TSL-based intermediate representation is generated first, from which both success and client-error scenarios are derived simultaneously from the OpenAPI Specification semantics, before any code is produced. This design grounds all test scenarios in the specification rather than in observed server behavior, making the generated suite independent of deployment state and enabling the LLM to reason about edge cases and business rules that might not surface through execution-guided exploration alone.

Finally, recent benchmarking efforts have started to systematize the evaluation of LLM-generated REST API tests. Kogler, Hangler, et al. (2026) proposed RESTestBench, a benchmark for evaluating the effectiveness of LLM-generated REST API test cases from natural-language requirements supported by OpenAPI Specifications. The benchmark distinguishes between non-refinement approaches, which generate tests from specifications or requirements without interacting with the SUT, and refinement-based approaches, which use execution feedback to iteratively improve generated tests. RESTestBench also

argues that traditional metrics, such as coverage and 5xx error counts, are insufficient to assess whether generated tests validate the intended functional behavior. To address this limitation, it introduces a requirements-oriented mutation testing strategy. This reinforces the importance of evaluating generated tests using multiple dimensions. Notably, RESTestBench includes RestTSLLM among the approaches under comparison, classifying it as a non-refinement approach that generates tests from the OpenAPI Specification without execution feedback — a classification consistent with the design choices described in this dissertation. The evaluation strategy adopted in this work — which includes execution success, coverage, mutation score, and a weighted composite score — is consistent with this broader movement toward multidimensional assessment of LLM-generated REST API tests.

Overall, the reviewed studies show that LLMs are being used in REST API testing through multiple strategies: enriching OpenAPI Specifications, mining constraints, generating realistic input values, identifying operation and parameter dependencies, guiding dynamic exploration, and producing executable test artifacts. However, many recent approaches rely on specialized models, fine-tuning, reinforcement learning, runtime feedback, RAG, or additional natural-language requirements. Comparatively less attention has been given to approaches that employ a structured intermediate representation to explicitly separate the reasoning about test scenarios from code generation — generating reusable integration tests from OpenAPI Specifications using general-purpose LLMs and prompt engineering alone, without fine-tuning, execution feedback, or additional requirements documentation. This gap motivates the present work, which investigates whether a structured TSL-based intermediate representation and decomposed prompting can support the generation of executable REST API integration tests in a reproducible, accessible, and model-agnostic manner.

3.4 Traditional Approaches without LLMs

Before the recent adoption of LLMs in REST API testing, several studies investigated traditional automated testing approaches based primarily on OpenAPI Specifications. Studies such as those by Golmohammadi, Zhang, and Arcuri (2023), Viglianisi, Dallago, and Ceccato (2020), and Corradini et al. (2021) analyze and compare black-box testing tools for automated REST API test generation, including RESTTestGen, bBOXRT, and RESTest. These tools typically derive test cases from the machine-readable elements of the specification, such as endpoints, HTTP methods, parameters, schemas, and expected

status codes.

Among these traditional tools, EvoMaster stands out as one of the most prominent representatives of *search-based* automated testing for REST APIs (ARCURI, 2019). Rather than parsing the OpenAPI Specification to derive test cases from its structural elements, EvoMaster applies evolutionary algorithms to automatically generate sequences of HTTP requests that maximize code coverage and fault detection. The tool operates in two modes: a black-box testing mode, which treats the SUT as an opaque service and relies solely on the OpenAPI Specification to guide test generation; and a white-box mode, which instruments the application's source code at runtime to collect execution feedback—such as branch distances and code coverage—and steer the evolutionary search toward unexplored code paths. The white-box mode consistently achieves higher structural coverage than the black-box alternative, at the cost of requiring direct access to, and instrumentation of, the SUT's source code (ARCURI, 2019; GOLMOHAMMADI; ZHANG; ARCURI, 2023).

EvoMaster and RestTSLLM address REST API testing from complementary perspectives. EvoMaster optimizes for structural coverage and the detection of runtime failures—crashes, unexpected status codes, and specification violations—through automated search, without attempting to interpret the semantic content of the specification. RestTSLLM, in contrast, operates exclusively in black-box mode and leverages LLMs to interpret the natural-language descriptions embedded in the OpenAPI Specification, producing tests that reflect the intended business logic and functional scenarios of each endpoint. The two approaches therefore differ not only in their underlying mechanism (search-based optimization vs. LLM-guided generation), but also in their primary objective: structural fault detection vs. specification-aligned functional validation.

Other traditional approaches have demonstrated relevant results in terms of coverage and fault detection, especially for robustness testing and the identification of server-side failures (GOLMOHAMMADI; ZHANG; ARCURI, 2023; VIGLIANISI; DALLAGO; CECCATO, 2020; CORRADINI et al., 2021). However, they often rely on predefined heuristics, schema-based validation, status-code oracles, and limited dependency inference mechanisms. As a result, they may struggle to generate semantically meaningful inputs, identify complex operation or parameter dependencies, and validate business rules that are described only in natural language within the OpenAPI Specification.

In this sense, traditional approaches provide an important baseline for automated REST API testing, but they also expose limitations that motivate the use of LLMs. Unlike these tools, LLMs can potentially interpret natural-language descriptions, infer implicit

constraints, and generate more context-aware test scenarios. Nevertheless, as discussed in the previous sections, LLM-based approaches also introduce new challenges related to reliability, reproducibility, evaluation, and the generation of executable and maintainable test artifacts.

Nevertheless, LLM-based approaches also introduce new challenges related to output reliability, evaluation methodology, model versioning, and — in more sophisticated architectures — the need for fine-tuning, runtime infrastructure, or execution feedback. These trade-offs are discussed throughout this chapter and acknowledged as limitations and directions for future work in this dissertation.

3.5 Research Gaps

Despite the rapid growth of research on LLM-based test generation, important gaps remain in the context of REST API integration testing. Existing studies show that LLMs can support different testing activities, including test input generation, unit test generation, constraint extraction, dependency inference, specification enrichment, dynamic exploration, and executable test generation. However, these approaches often target different objectives or rely on additional mechanisms such as fine-tuned models, reinforcement learning, runtime feedback, retrieval-augmented generation, explicit natural-language requirements, or specialized dependency graphs.

In the specific context of REST API testing, recent approaches have demonstrated important advances, but they still leave room for further investigation. Some works (KIM; STENNETT; SHAH, et al., 2024) focus on enriching OpenAPI Specification documents rather than generating complete executable tests; others optimize dynamic exploration and fault detection, but do not produce reusable test artifacts that can be easily versioned and integrated into development workflows. More recent approaches generate executable integration tests, but frequently depend on explicit business requirements, execution feedback from the SUT, or more complex agent-based and refinement-based architectures (KIM; SINHA; ORSO, 2025; KIM; STENNETT; SINHA, et al., 2025). Therefore, there is still limited evidence on how effective general-purpose LLMs can be when guided only by prompt engineering and information available in the OpenAPI Specification.

Another relevant gap concerns the evaluation of generated tests. Traditional studies often emphasize coverage, status codes, or fault detection, while recent benchmarks argue that these metrics alone may be insufficient to assess whether generated tests capture

meaningful functional behavior (KOGLER; HANGLER, et al., 2026). This motivates the use of broader evaluation strategies that consider not only whether tests execute successfully, but also whether they improve structural coverage, reveal behavioral faults, and provide useful evidence of test effectiveness.

In this context, this work investigates the automated generation of reusable REST API integration tests from OpenAPI Specification documents using general-purpose LLMs, decomposed prompting, and a TSL-based intermediate representation. The study contributes empirical evidence by comparing multiple LLMs and evaluating the generated tests through complementary metrics, including success rate, coverage, mutation score, and a weighted composite score. By doing so, it provides a broader understanding of the potential and limitations of LLMs for automating the production of executable integration tests for REST APIs.

4 The RestTSLLM Approach

RestTSLLM is an approach designed to address the challenges of REST API testing, particularly in generating scenarios, input data, and executable tests. The approach is based on the premise of being easily replicable, without requiring fine-tuning or specialized models, relying solely on prompt engineering and the use of the *few-shot* and *decomposed prompting* techniques. The contents of all prompts is presented in Section 5.4. Another essential premise is that its structure is easily adaptable and extensible to other types of tests that face similar challenges and other technologies. This chapter aims to detail the approach proposed in this dissertation, which is visually summarized in Figure 3.

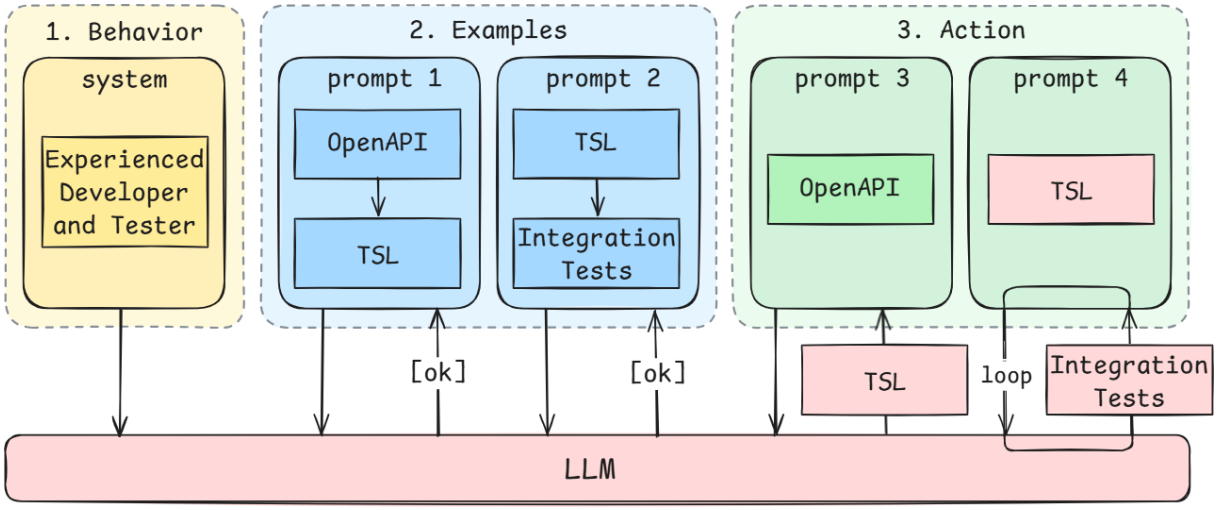

Figure 3: Overview of RestTSLLM approach

The proposed prompt engineering is organized into three main stages: (1) Behavior – defining the behavior of the LLM, (2) Examples – presenting examples, and finally, (3) Action – requesting the execution of the task.

The diagram in Figure 3 adopts a color-coded structure to distinguish different components and roles within the RestTSLLM approach visually. The three main stages are highlighted in distinct colors: yellow for Behavior, blue for Examples, and green for Action. The LLM and all outputs generated by it are shown in red. It is important to

note that in *prompt 4*, the TSL input is shown in red, to indicate that it is composed of the output previously generated by the LLM in *prompt 3*.

4.1 Stage I - Behavior

In the first stage, **Behavior**, the LLM is instructed, through the *system prompt* parameter, to act as an experienced developer and tester, capable of understanding REST API specifications and generating coherent and functional integration test artifacts. This sets the expected behavior for the model in the following stages. It also allows for restricting or enabling functions, guiding the conversational flow, imposing ethical or safety constraints, and preparing the response environment, among other possibilities that limit or direct its behavior. The use of this technique allows us to improve the performance and generalization of the generated result, ensuring greater alignment with the expectations of how the generation of results will be done given the established context (MU et al., 2025; ZHANG, L. et al., 2024; NEBULY, 2024).

In addition to acting as an experienced developer and tester, the model is instructed to operate as if it has a strong knowledge of the target technology used for test generation, of rule extraction from OpenAPI Specification and conversion to TSL using the Category–Partition method (OSTRAND; BALCER, 1988), and of testing best practices such as organizing tests in the AAA pattern (Arrange, Act, Assert), boundary testing, equivalence classes, and scenario design aiming for high coverage. Furthermore, the model's behavior is configured to return only the requested code, without textual explanations or justifications.

4.2 Stage II - Examples and Rationale for TSL

The second stage, **Examples**, consists of two steps that apply the *few-shot* and *decomposed prompting* techniques (KHOT et al., 2023; SADOWSKI; CHUDZIAK, 2026) to guide the model's learning through concrete examples: **Prompt 1:** An OpenAPI Specification is presented as input, and the desired output is a set of test cases written in Test Specification Language (TSL), whose main objective is to provide a formal, and standardized language for the specification, and documentation of software tests, usually formatted in *yaml*. TSL is designed to help define structured test cases, including scenarios, inputs, expected outputs, and conditions, in a clear, and understandable way, promoting greater accuracy, reusability, and automation in the testing process (OSTRAND; BALCER, 1988). This

stage teaches the LLM how to interpret OpenAPI Specification documents and extract relevant information to define structured test scenarios; **Prompt 2:** The TSL presented in the first step is now used as input, and the expected output is a set of executable integration tests. The approach supports the generation of tests in any technology—depending on what is presented in the model. This step teaches the LLM how to translate abstract test cases into functional code based on the structure and conventions of the chosen technology while following integration testing best practices.

This intermediate generation step using TSL was designed to improve the LLM's effectiveness. By separating the test case reasoning from the implementation, the first prompt allows the model to concentrate exclusively on understanding the business rules described in the API specification, without being burdened by concerns related to code structure or syntax. Once the test scenarios are clearly defined in TSL, the second prompt focuses solely on converting those predefined cases into functional test code. This separation of concerns helps the LLM perform more efficiently in each task, reducing complexity and improving the output quality (KHOT et al., 2023; SADOWSKI; CHUDZIAK, 2026).

4.3 Stage III - Action and Practical Constraints

The third stage, **Action**, also contains two steps in which the LLM applies the logic learned from the examples to new inputs. Here, the model generates the outputs: **Prompt 3:** A real OpenAPI Specification—different from the one used in the examples—is provided to the LLM. Based on what it learned in *prompt 1*, the model generates a new TSL, describing the scenarios that should be tested; **Prompt 4:** The TSL generated in the previous step is then used as input. Following the logic of *prompt 2*, the model produces the final integration test code. These tests are ready to be executed according to the framework and language defined in the example prompts provided to the LLM.

During the experiments, we observed that some LLMs — particularly those with lower output token limits—had difficulty generating all the tests in a single response. This often resulted in truncated outputs. To address this, we implemented a strategy where *prompt 4* is executed in a loop, requesting the generation of subsets of tests grouped by specific tags or methods defined in the OpenAPI Specification. These tags are carried over to the TSL to segment the test cases, allowing the LLM to produce valid output in manageable parts while preserving the overall structure and completeness of the test suite.

4.4 Illustrative Example (TSL and Code)

Listing 1, Listing 2, and Listing 3 illustrate the three artifacts involved in the RestTSLLM pipeline for a single endpoint of the *hotels-api* project, using Deepseek R1 as the LLM. Listing 1 shows the relevant fragment of the existing OpenAPI Specification of the *hotels-api* project that was provided as input to the approach — only the path and schemas directly involved in the example are shown, as the full specification includes additional endpoints and components. From this input, Deepseek R1 produced the outputs shown in the following listings. Listing 2 shows the TSL generated in response to prompt 3 (action to generate TSL), which defines the scenario *"Login Valid Credentials Returns Token"* in a structured and declarative format. This format allows the model to describe the endpoint to be tested, required preconditions, input data, and expected output. Listing 3 presents the integration test code generated from that TSL in response to prompt 4 (action to generate integration tests), using xUnit (.NET). It includes all the necessary steps to perform the test according to the defined expectations. In general, the input data defined in the TSL is preserved during test generation, although minor adjustments may be applied to ensure repeatability. In the example shown in Listing 3, for instance, a function was used to dynamically generate a unique email address, replacing the static value originally defined in the TSL. This strategy prevents failures in subsequent test executions that could occur due to uniqueness constraints in the SUT, as taught in the examples presented to the LLM.

OpenAPI Specification Fragment (hotels-api)

```
{
  "paths": {
    "/api/accounts/tokens": {
      "post": {
        "tags": ["Account"],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": { "$ref": "#/components/schemas/LoginDto" }
            }
          }
        },
        "responses": {
          "200": {
            "description": "OK",
            "content": {
              "application/json": {
                "schema": { "$ref": "#/components/schemas/AuthResponseDto" }
              }
            }
          }
        }
      }
    }
  },
  "components": {
    "schemas": {
      "LoginDto": {
        "required": ["email", "password"],
        "type": "object",
        "properties": {
          "email":    { "type": "string", "format": "email", "minLength": 1 },
          "password": { "type": "string", "minLength": 1 }
        }
      },
      "AuthResponseDto": {
        "required": ["userId", "token", "refreshToken"],
        "type": "object",
        "properties": {
          "userId":       { "type": "string", "minLength": 1 },
          "token":        { "type": "string", "minLength": 1 },
          "refreshToken": { "type": "string", "minLength": 1 }
        }
      }
    }
  }
}
```

Listing 1: Relevant OAS fragment of the *hotels-api* project used as input to the approach

Generated TSL

```yaml
- id: TC101
  group: Account
  name: Login Valid Credentials Returns Token
  endpoint: /api/accounts/tokens
  method: POST
  preconditions:
    - "User with email 'valid@test.com' exists"
  request_body:
    email: "valid@test.com"
    password: "Val1d!Pass"
  expected_response:
    status_code: 200
    body:
      userId: is string not empty
      token: is string not empty
      refreshToken: is string not empty
```

Listing 2: TSL generated by Deepseek R1 from the *hotels-api* OAS fragment

Generated Integration Tests

```csharp
[Fact]
public async Task TC101_Login_Valid_Credentials_Returns_Token()
{
    // Arrange
    var email = GenerateUniqueEmail();
    var password = "Val1d!Pass";

    // Create user first
    await CreateUserAsync(new
    {
        firstName = "John",
        lastName = "Doe",
        email,
        password,
        isAdmin = false
    });

    // Act
    var response = await LoginAsync(email, password);

    // Assert
    var body = await response.Content.ReadFromJsonAsync<JsonObject>();
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
    Assert.False(string.IsNullOrEmpty(body["userId"].ToString()));
    Assert.False(string.IsNullOrEmpty(body["token"].ToString()));
    Assert.False(string.IsNullOrEmpty(body["refreshToken"].ToString()));
}
```

Listing 3: Integration test generated by Deepseek R1 from the TSL in Listing 2

5 Experimental Evaluation

This dissertation conducts two complementary evaluations to validate the RestTSLLM approach, both motivated by the research questions. The first, presented in this chapter, is a controlled experiment designed to answer **RQ1** (*Are LLMs effective in generating integration tests that reflect the intended business logic and context?*) and **RQ2** (*Which LLM is the most effective in generating integration tests?*). In this experiment, a set of LLMs was applied to generate integration tests for real-world REST API projects using the RestTSLLM pipeline, and the generated tests were evaluated against objective quality metrics. The second evaluation, an acceptance survey targeting practitioners involved in the REST API development lifecycle, addresses **RQ3** and is presented in Chapter 6.

Figure 4 illustrates the methodological steps of the controlled experiment. Each step is summarized below, with further details presented in the following sections.

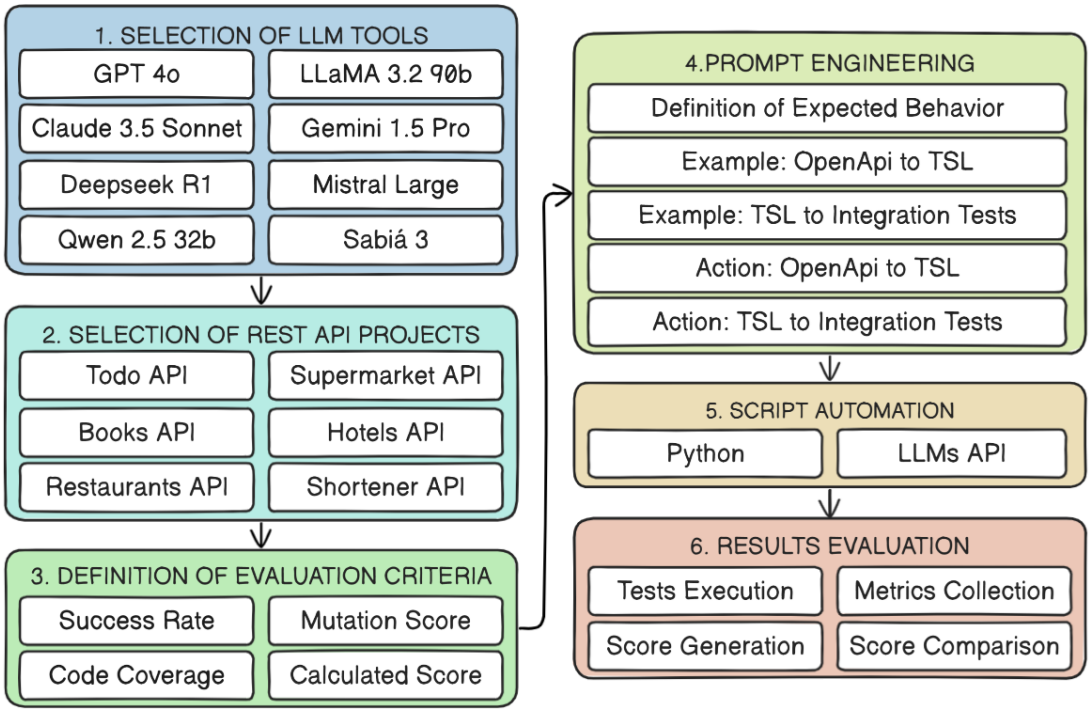

Figure 4: Experimental evaluation flow

The first step, **Selection of LLM tools**, involved choosing a diverse set of language models to support the evaluation of our proposed approach. We selected GPT 4o (OpenAI), LLaMA 3.2 90b (Meta), Claude 3.5 Sonnet (Anthropic), Gemini 1.5 Pro (Google), Deepseek R1 (Deepseek), Mistral Large (Mistral), Qwen 2.5 32b (Alibaba), and Sabiá 3 (Maritaca), which stand out in popularity, and performance in code generation tasks. We also included Sabiá 3 (Maritaca), a Brazilian LLM designed to support Portuguese (Section 5.1).

The second step, **Selection of REST API Projects**, consisted of selecting six REST API repositories that met the following criteria: authorized for use in this study, contained an OpenAPI Specification, were relevant on GitHub, had at least one integration with a database or external service, and developed using the target technology of the experiment (Section 5.2).

The third step, **Definition of Evaluation Criteria**, involved defining the set of metrics to assess the effectiveness of the integration tests generated by the LLM tools. For this purpose, we selected the success rate, coverage, and mutation score to compute a calculated score from these metrics and evaluate the performance of LLMs (Section 5.3).

The fourth step, **Prompt Engineering**, expanded on the experiment details in creating prompts used to guide the LLMs in RestTSLLM approach (Section 5.4).

The fifth step, **Script Automation**, referred to the development of an automated script for integrating with the LLMs to process the prompts, and their results (Section 5.5).

Finally, the sixth step, **Results Evaluation**, described the evaluation of the integration tests generated by the LLMs (Section 5.6).

5.1 Selection of LLM Tools

As mentioned earlier, the LLM tools selected for this study include GPT 4o (OpenAI), LLaMA 3.2 90b (Meta), Claude 3.5 Sonnet (Anthropic), Gemini 1.5 Pro (Google), Deepseek R1 (Deepseek), Mistral Large (Mistral), Qwen 2.5 32b (Alibaba), and Sabiá 3 (Maritaca). The selection criteria were based on the popularity of LLMs, measured as the number of sources — online benchmarks and curated model rankings/listings (OPENROUTER, 2025; VELLUM, 2025; UNITE.AI, 2025; DEFINITION, 2025; ZAPIER, 2025; LIVEBENCH, 2024; SHAKUDO, 2025; BOTPRESS, 2025), and prior academic studies (XIA; DENG; ZHANG, 2024; AYDIN et al., 2025; QIU et al., 2024; HAGOS; BATTLE;

RAWAT, 2024) — in which each model appears. Appendix A details this cross-reference: Table 11 lists the selected models with their popularity scores (total mentions across all sources), and Table 12 lists models that appeared in at least two sources but were excluded from the study. Additionally, only models with publicly available documentation, and the ability to be used both via API, and user interface were considered. The versions listed reflect those available during the first half of 2025, when the experiment was conducted. Given the rapid pace of LLM development, newer releases may have become available since then; this is acknowledged as a limitation and discussed further in Section 7.4 and as future work in Chapter 8.

Table 1: Selected LLMs

Model	**Version**	**Company**	**Country**	**License**
GPT	4o	OpenAI	USA	Private
LLaMA	3.2 90b	Meta	USA	Open-source
Claude	3.5 Sonnet	Anthropic	USA	Private
Gemini	1.5 Pro	Google	USA	Private
Deepseek	R1	Deepseek	China	Open-source
Mistral	Large	Mistral	France	Open-source
Qwen	2.5 32b	Alibaba	China	Open-source
Sabiá	3	Maritaca	Brazil	Private

From this list, we selected the seven most popular models. In addition to these, we included one more LLM to ensure the presence of a model with higher accuracy in interpreting prompts in Portuguese — the main language in the prompts in this study.

We chose Sabiá 3 (Maritaca), one of the most visible Brazilian LLM families for Portuguese, supported by peer-reviewed technical reports and public availability via API integrations (ABONIZIO et al., 2024; FREITAG; GOIS, 2024; PIRES et al., 2023; ALMEIDA et al., 2024).

As Sabiá 3 was developed in Brazil, the author's home country, its inclusion also contributes to highlighting Brazilian research and technological development within the scientific community.

The final list of the selected LLMs is available in Table 1, which indicates the selected models, the version of each model used (the most recent version available at the time of selection was considered), the provider company, its origin country, and whether the model is available as open-source or only through paid versions.

5.2 Selection of REST API Projects

The criteria used for selecting REST API projects were defined to meet three goals: ensuring technical compatibility with the RestTSLLM pipeline, guaranteeing the scientific validity of the selected projects, and maintaining the practical feasibility of the experiment. A project was considered eligible if it met all of the following: having a formal or permissive license that authorizes free use in this study; containing at least one REST API application, which is the focus of the research; providing an OpenAPI Specification to serve as a basis for generating integration tests; demonstrating relevance on GitHub, with a combined number of stars and forks greater than 100, to avoid non-functional or unvalidated projects; including at least one integration with a database or another API to ensure some level of integration testing; offering documentation with installation, configuration, and usage guides to support comprehension and reproducibility; having a limited number of endpoints to constrain the resources required for the experiment; and being primarily implemented in recent versions of .NET — a technology supported by a leading software company (Microsoft), deeply mastered by the author, and widely adopted in the industry for REST API development (FORTUNE BUSINESS INSIGHTS, 2023).

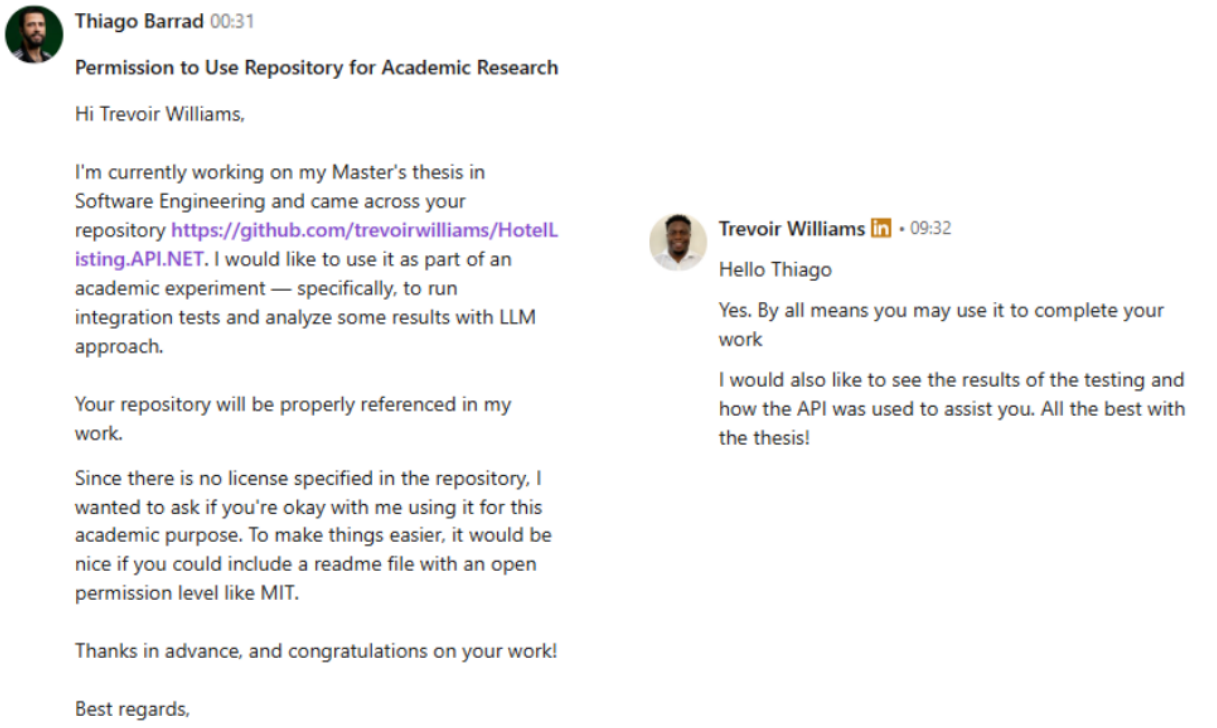

Figure 5: Formal authorization for the use of the *hotels-api* project

After searching on GitHub for projects that met the defined criteria, we selected six .NET-based REST API projects with at least one integration, an OpenAPI Specification,

and documentation in a *README.md* file. The selected projects were *todo-api* (FOWLER, 2025), *supermarket-api* (GOMES, 2025), *books-api* (SOYSA, 2025), *hotels-api*, (WILLIAMS, 2025), *restaurants-api* (KOZERA, 2025), and *shortener-api* (ERDAL, 2025). Projects and selection criteria are presented in Table 2, the number of endpoints per method is shown in Table 3, and the dependency map is presented in Table 4, which illustrates the components considered as external integrations in integration tests. Most had an MIT License[1], allowing free use, while formal permission was obtained for the *hotels-api* project as shown in Figure 5. We also included a CLOC[2] (Count Lines of Code) column to indicate the size of each project based on the number of lines of C# code. For simplification purposes, in Table 2, an acronym was used to represent some columns: *DV* → .NET version; *E* → total of endpoints; *D* → total of dependencies, *S* → total of stars; *F* → total of forks; and *SF* → Sum of stars and forks.

Table 2: Selected REST API projects and their evaluation criteria data

Project	**License**	**DV**	**E**	**D**	**S**	**F**	**SF**	**CLOC**
todo-api	MIT	7.0	7	1	2.964	441	3.405	1.576
supermarket-api	MIT	8.0	8	1	484	166	550	977
books-api	MIT	8.0	5	2	145	92	237	809
hotels-api	Formal	7.0	14	1	76	42	118	2.843
restaurants-api	MIT	8.0	10	2	64	39	103	1.804
shortener-api	MIT	9.0	3	2	78	22	100	1.055

Table 3: Selected REST API projects — endpoint breakdown by HTTP method

Project	**#GET**	**#POST**	**#PUT**	**#PATCH**	**#DELETE**	**Total**
todo-api	2	3	1	0	1	7
supermarket-api	2	2	2	0	2	8
books-api	2	1	1	0	1	5
hotels-api	5	5	2	0	2	14
restaurants-api	4	3	0	1	2	10
shortener-api	2	1	0	0	0	3
Total	**17**	**15**	**6**	**1**	**8**	**47**

[1]Official website of the MIT license: https://opensource.org/license/mit
[2]Project used to count only C# lines with CLOC: https://github.com/AlDanial/cloc

Table 4: Dependency map and components of selected projects

Project	SQLite	PostgreSQL	SQL Server	Azure Storage	MongoDB	Redis
todo-api	✓	✗	✗	✗	✗	✗
supermarket-api	✓	✗	✗	✗	✗	✗
books-api	✗	✓	✗	✗	✗	✓
hotels-api	✗	✗	✓	✗	✗	✗
restaurants-api	✗	✗	✓	✓	✗	✗
shortener-api	✗	✗	✗	✗	✓	✓

Among the selected projects, only one had pre-existing integration tests, which limited the possibility of directly comparing the results generated by LLMs with prior test artifacts. In the Chapter 7 (Results and Discussion) we discuss how we mitigated this issue to guarantee that the possible prior knowledge of these pre-existing tests of the only project has not affected the results of this experiment. However, this also reinforces the practical relevance of approaches that simplify test creation, as it reflects the common scenario where most real-world projects lack integration test. Additionally, the absence of prior tests in most APIs reduces the likelihood of overlap or memorization effects by the evaluated models, supporting a more unbiased assessment of their capabilities.

5.3 Definition of Evaluation Criteria

We adopted test success rate, coverage, and mutation score as evaluation criteria. The success rate is an immediate indicator of the system stability for the executed set of integration tests, reflecting whether any failures were observed in the evaluated scenarios. However, the isolated analysis of this metric is insufficient to assess test quality, making it necessary to use complementary metrics (JAIN et al., 2023).

Coverage is a widely used metric in software testing, frequently reported alongside the success rate. It quantifies the proportion of the software exercised during test execution, typically at the level of statements or branches. In this study, we focus on branch coverage, as it offers a more precise assessment of control-flow exploration by checking whether all decision outcomes are exercised. Higher coverage tends to indicate more comprehensive evaluation and may increase the likelihood of exposing failures within the SUT (DELAMARO; JINO; MALDONADO, 2013; OUÉDRAOGO et al., 2024).

The mutation score, in turn, is considered a stricter metric than coverage, although

complementary to it. It assesses the sensitivity of a test suite to small changes in the system's logic by introducing artificial faults (mutants) and verifying whether the tests detect them. A high mutation score indicates that the tests effectively identify behavioral deviations, suggesting a stronger alignment with the intended functional requirements (ZHANG, P. et al., 2022).

Thus, the combination of these metrics was adopted as the evaluation criterion in this study, as it allows a comprehensive analysis of key aspects of software testing quality: ensuring that the system behaves as expected, guaranteeing a significant amount of coverage, and effectiveness of the test suite in detecting subtle logic faults (JAIN et al., 2023; ZHANG, P. et al., 2022).

To compute the *Calculated Score*, we used a weighted score based on the three evaluation metrics adopted in this study: success rate, coverage, and mutation score. This score was designed to summarize the overall effectiveness of the generated test suites by combining their ability to execute successfully, exercise the application code, and detect injected faults. Since all metrics are expressed as percentages within the same range, no normalization was required before aggregation. Given that no metric was considered more important than the others, equal weights of 33.33% were assigned to each metric.

The formula used to calculate the *Calculated Score*, denoted by S, is presented below:

$$S = w \cdot SuccessRate + w \cdot Coverage + w \cdot MutationScore$$

The formula uses the previously mentioned metrics: success rate ($SuccessRate$), coverage ($Coverage$), and mutation score ($MutationScore$), where $w = 33.33\%$. Thus, the *Calculated Score* corresponds to the arithmetic mean of the three metrics, giving equal importance to execution success, structural coverage, and fault-detection effectiveness.

To compare the effectiveness of the LLMs, we used the average calculated score as the final score of their performance.

These evaluation criteria were designed to address part of the research questions proposed in this study. We combined quantitative and qualitative evaluation strategies to answer the proposed research questions fully. We performed a manual and subjective analysis for **RQ1**, which investigates whether LLMs can generate integration tests aligned with the intended logic and context. All test cases and generated code were reviewed by the author to assess their coherence with the described business rules, the clarity of the scenarios, and their correspondence with the structure defined by the OpenAPI

Specification and the expected test patterns. The effectiveness aspect of **RQ1**, as well as the full assessment required by **RQ2**, was addressed through the objective metrics described above. The average calculated score derived from these metrics enabled us to rank the models according to their overall performance and identify which LLMs were the most effective.

5.4 Prompt Engineering

In addition to the general aspects of the RestTSLLM approach previously explained in Chapter 4, this section explains the complementary peculiarities of applying the approach in our experimental evaluation.

As the target technology for generating REST API integration tests, we chose .NET with xUnit (MICROSOFT, 2025), given the author's depth and prior experience with the technology, supported by a leading software company (Microsoft), and widely adopted in the industry for developing and testing REST APIs (FORTUNE BUSINESS INSIGHTS, 2023). Given this choice, for our experimental evaluation, the specific examples presented in *prompt 2* — converting TSL to integration tests — provided examples of tests written in .NET with xUnit. Consequently, in *prompt 4*, the tests generated for the inputs of the previously selected real projects were also produced by the model in this technology.

The content of the prompts was written and executed in English and Portuguese while the OpenAPI Specifications remained in English. We observed that the performance was equivalent and suitable when testing both fully-English and partially-Portuguese prompts, given that the smallest part of the prompt is textual instructions and the predominant content is examples and codes. We chose to keep the prompts in Portuguese, the native language of the author's home country, aiming to support analyses focused on the local language, contribute to technological development, and highlight Brazilian research within the scientific community.

The RestTSLLM approach employs five prompt types arranged in a fixed conversational sequence. Listing 4 defines the system prompt that establishes the model's persona and output constraints. Listings 5 and 6 provide few-shot examples that teach the model the expected conversion patterns — from OAS to TSL and from TSL to xUnit tests, respectively. Listing 7 triggers the actual TSL generation for the target API, and Listings 8 and 9 drive the iterative generation of integration tests, one endpoint group at a time.

The generic and full prompts used in this study are available on GitHub[3]. Below we illustrate the prompts, omitting extensive code and example sections.

System Command

You are a highly experienced systems analyst with in-depth knowledge of software development in DotNet and integration testing with xUnit.

You know how to extract rules to create good tests from OpenAPI specifications and convert them to TSL (Test Specification Language) using the Category-Partition method.

You consider high coverage important, as well as the application of bounds and equivalence class testing.

You develop integration tests using the AAA standard (act, arrange, assert).

You only respond to code or files, without responding to comments or any other additional text that would hinder the automation of converting the response into functional code.

Listing 4: System prompt: Persona and behavioral constraints.

Prompt 1 – OpenAPI to TSL

I will teach you best practices for converting an OpenAPI specification to TSL:

Example OpenAPI specification:

```
/// ...  full example of OpenAPI (json)
```

Example of TSL generated from the OpenAPI specification:

```
/// ...  full example of TSL with all test cases (yaml)
```

Just reply "OK" if you understand.

Listing 5: Prompt 1: Few-shot example prompt: OpenAPI specification to TSL conversion.

[3] https://github.com/uffsoftwaretesting/RestTSLLM/blob/main/general-files/default-english-prompts/files.md

Prompt 2 – TSL to Integration Tests

Now I will teach you best practices for converting a TSL to integration tests with dotnet (xUnit). I will also always provide the namespace to use.

Given the TSL provided above and the `<project_name>.IntegrationTests` namespace, a conversion to integration tests with dotnet (xUnit) should generate:

```
/// ...  full example of integration tests (.NET xUnit)
```

Just reply "OK" if you understand.

Listing 6: Prompt 2: Few-shot example prompt: TSL to xUnit integration tests conversion.

Prompt 3 – Convert real OpenAPI to TSL

Given the example I gave you of how to convert an OpenAPI specification to TSL, I need you to provide me with the TSL in yaml format for the following OpenAPI specification.

It is important to emphasize the importance and necessity of generating as many test scenarios as possible to maximize coverage.

You should generate all cases without returning a response asking me to add anything.

```
/// ...  full real OpenAPI (json)
```

Listing 7: Prompt 3: Generate TSL from the target OpenAPI specification.

Prompt 4.1 – Convert generated TSL to Integration Tests (first endpoint group)

Very good! Given the generated TSL, now following the example I gave earlier, I would like you to generate the integration tests with dotnet (xUnit) for the TSL returned, considering the `<project_name>.IntegrationTests` namespace.

It is important to emphasize that you should not reuse elements created in one test in another; for example, you should always create a new user with a different username for each test. Limit validations must be efficient. If a property accepts 10 to 30 characters, for example, the limit test should at least test with null, empty (''), and lengths of 9, 10, 30, and 31.

You should never assume an "id" exists or ask me to provide it. The test must create all the resources it needs by calling the API. This is mandatory.

You must generate all tests for all scenarios generated in the TSL, without omitting any, without returning a response asking me to add anything.

Generate integration tests for all possible scenarios. First, generate integration tests for all scenarios in `group: <group_name>`.

Listing 8: Prompt 4.1: generate xUnit tests for the first TSL endpoint group.

Prompt 2.N – Convert generated TSL to Integration Tests (remaining groups)

Thank you!

It is important to emphasize again that you should not reuse elements created in one test in another. For example, you should always create a new user with a different username for each test.

You should generate all tests for all scenarios generated in TSL, without omitting any, and without returning a response asking me to add anything.

Now, generate integration tests for all scenarios in `group <group_name>`.

Listing 9: Prompt 4.N: Generate xUnit tests for each subsequent TSL endpoint group (repeated for each group).

5.5 Script Automation

Studies involving LLMs commonly apply prompt engineering in their experiments using the interface provided by the model developer (WANG et al., 2024). However, this approach can become complex when executing the same sequence of prompts and collecting their results across a larger set of prompts and LLMs. For our study, we developed a Python script capable of integrating with the studied LLMs, allowing easy connection to additional models for future research, and reuse of the code.

The developed Python script, also called *llm-processor*, automates the entire process: loading prompt files, handling model configurations, executing each step while maintaining conversational context, and saving the outputs and metrics. Figure 6 illustrates this flow: the script loads the model configurations from `llms_config.json` and the prompt files from the `prompts/` directory, then iterates over each LLM and each prompt, sending requests to the respective remote API, receiving the results, and persisting both the generated outputs and the execution metrics to the `outputs/` directory. This made it possible to run consistent, large-scale experiments efficiently across different LLMs. All selected LLMs were accessed via remote APIs; none were executed locally.

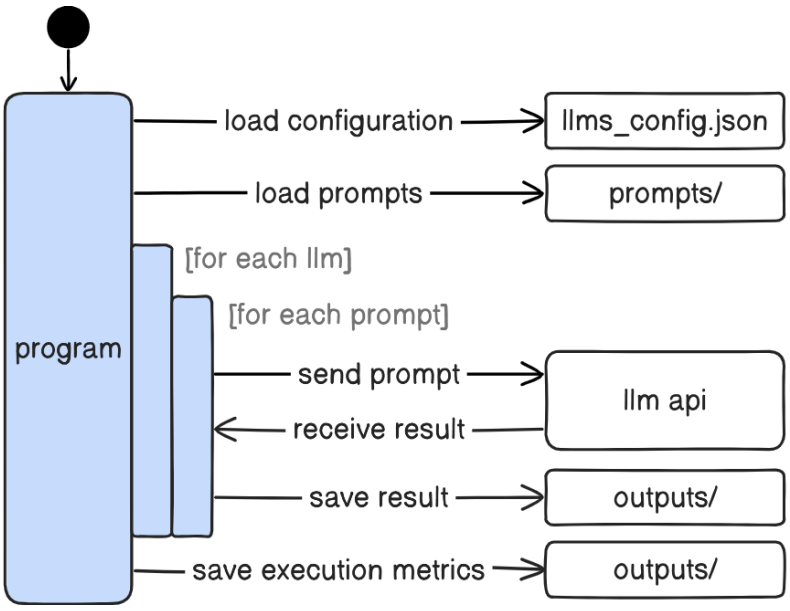

Figure 6: Simplified automated script flow

The source code of the automated script is available on GitHub[4].

[4] https://github.com/uffsoftwaretesting/RestTSLLM/tree/main/llm-processor/README.md

5.6 Execution and Results Collection

The final execution that resulted in the outcomes of this study took place on March 9, 2025, and lasted 2 hours, and 30 minutes to process all prompts across all LLMs. The execution was done on a computer with an i7 processor and 32 GB RAM.

The results of the prompt executions in the LLMs can be accessed on GitHub[5].

After generating integration tests by LLMs in xUnit with .NET, the target project was duplicated for each LLM, the tests were manually copied, and then executed with the collection of the metrics described in Section 5.3.

To obtain the success rate and coverage results, we ran the tests using the Visual Studio 2022 tool (MICROSOFT, 2025). We used the branch coverage metric previously explained in Section 5.3.

To obtain the mutation score we used the Stryker.NET tool (STRYKER, 2025) with maximum mutation level enabled (*Advanced*), creating mutants for regex, string literals, collection initializer, statements like block, checked, and assignment, and operators like arithmetic, logical, bitwise, equality, boolean, update, and unary, and methods like string, math, and linq (STRYKER, 2025). We disabled the option to combine mutants in the same execution for greater assertiveness of the result, even though this option results in a longer execution time, given that mixed mutants can present unwanted side effects (STRYKER, 2025).

The metrics collected from the test executions, and the evidence of these executions are provided in Appendix B (Tables 13, 14, 15, 16, 17, and 18) and in a public PDF file on our GitHub[6].

[5] https://github.com/uffsoftwaretesting/RestTSLLM/tree/main/projects/files.md
[6] https://github.com/uffsoftwaretesting/RestTSLLM/tree/main/pdfs/all_results.pdf

6 Acceptance Survey

The experimental evaluation presented in Chapter 5 assessed the technical effectiveness of the RestTSLLM approach by measuring success rate, coverage, and mutation score across eight LLMs and six REST APIs. Those metrics, however, capture only one dimension of a technology's viability: they tell us *how well* the approach works in a controlled setting, but not *whether practitioners would actually adopt it* in real-world contexts — since performance gains are often obstructed by unwillingness to accept and use available systems (DAVIS, 1989).

To address this gap, this chapter describes the methodology of an acceptance survey designed to answer the following research question:

RQ3: *What is the level of acceptance of the RestTSLLM approach among practitioners involved in the REST API development lifecycle?*

The survey investigated the perception of the approach's utility for practitioners, their intention to adopt it, and their suggestions for improvement.

6.1 Setting Objectives

This study adopted an *opinion survey* as its research method. An opinion survey gathers knowledge from individuals to understand specific aspects of a population (WOHLIN et al., 2012), making it particularly suited for capturing perceptions, attitudes, and adoption intentions that cannot be directly measured through technical experimentation.

The goal of this survey, stated using the Goal–Question–Metric (GQM) template (BASILI; CALDIERA; ROMBACH, 1994), was:

> **Analyze** the RestTSLLM approach **with the purpose of** evaluating its acceptance **with respect to** perceived utility, intention of use, and applicability

in real-world scenarios **from the point of view of** practitioners involved in the REST API development lifecycle **in the context of** software development in industry, academia, and related domains.

The primary research question this survey aimed to answer was **RQ3**, as stated above. To operationalize it, we decomposed RQ3 into the following sub-questions:

- **RQ3.1:** How do practitioners perceive the utility of RestTSLLM for automating integration test generation?
- **RQ3.2:** To what extent do practitioners trust and intend to use a tool implementing the RestTSLLM approach?
- **RQ3.3:** Do practitioners consider the two-step pipeline (OAS → TSL → tests) a useful design decision?
- **RQ3.4:** What integration modalities do practitioners prefer for such an approach?
- **RQ3.5:** Do acceptance levels differ across professional profiles (role, experience, and technical background)?

6.2 Study Design

We conducted this study as a *structured questionnaire survey*, following the opinion survey guidelines described by Kitchenham and Pfleeger (2008) and aligned with the ACM SIGSOFT Empirical Standards for Questionnaire Surveys (RALPH et al., 2020; ACM SIGSOFT EMPIRICAL STANDARDS, 2020). This methodological choice was justified on three grounds:

- **Breadth of coverage:** A questionnaire enables the collection of data from a geographically distributed population of practitioners in a time-efficient manner, maximizing the diversity of respondents and the generalizability of results.
- **Quantitative measurement:** A Likert scale-based instrument allows systematic quantitative analysis of perceptions and attitudes (JOSHI et al., 2015).
- **Anonymity:** The anonymous format encourages honest responses, particularly on evaluative items concerning the novelty and perceived risks of using AI-generated tests in production.

The questionnaire was structured in three main blocks: (i) participant profiling, (ii) adoption and usage intention, and (iii) open-ended suggestions. Block (ii) required participants to read a brief description of the RestTSLLM approach, inspect illustrative examples of the pipeline inputs and outputs, and answer Likert-scale items. An eliminatory comprehension check question was embedded to ensure data quality, as detailed in Section 6.3.

6.2.1 Target Population and Sampling Strategy

The target population of this survey consisted of professionals involved in the REST API development lifecycle, including software engineers, quality assurance engineers and testers, technical leads and managers, software architects, DevOps/SRE practitioners, and researchers, educators or students in software engineering area. This population was appropriate because the RestTSLLM approach directly addresses challenges faced in this community, and their acceptance is a prerequisite for practical adoption.

Given the absence of a comprehensive public registry of such professionals, this study adopted a *non-probabilistic purposive sampling* strategy (RALPH et al., 2020; ACM SIGSOFT EMPIRICAL STANDARDS, 2020), targeting individuals who met at least one of the following criteria: (i) participation in the REST API development cycle; (ii) experience with integration test or quality assurance; (iii) familiarity with OAS, Swagger, Postman, or similar tools; or (iv) knowledge of or interest in LLMs applied to software development.

No minimum total sample size was formally established a priori, which is consistent with non-probabilistic purposive sampling strategies commonly adopted in exploratory opinion surveys (RALPH et al., 2020; ACM SIGSOFT EMPIRICAL STANDARDS, 2020). No quotas were defined per role or background group; instead, the goal was to maximize overall response diversity. The demographic data collected in Block 1 enabled post-hoc stratification, allowing results to be reported disaggregated by role, experience level, and technical background.

6.3 Survey Instrument

The questionnaire consisted of 17 questions distributed across three blocks: eight profiling questions in Block 1 (Q1.1–Q1.8), seven questions in Block 2 (one eliminatory check and six Likert-scale items, plus one multiple-choice item on integration modalities), and two optional open-ended questions in Block 3. The complete instrument is described below,

and the full version, with all instructions, is presented in the Appendix D.

6.3.1 Block 1 — Participant Profile

Block 1 collected characterization data about participants along three dimensions: their professional *role*, their *experience*, and their technical *background*. These dimensions enabled the sample to be characterized and the responses to be stratified during analysis (Section 6.6.1). Table 5 summarizes the eight profiling questions and the dimension each one addresses: role was captured by Q1.1; experience by Q1.3 (years in the field) and by Q1.5–Q1.8 (hands-on experience with specific practices and technologies); and background by Q1.2 (declared knowledge areas) and Q1.4 (technical stack).

Table 5: Block 1 — Participant profiling questions

ID	Question	Type	Dimension
Q1.1	Current role	Single choice	Role
Q1.2	Knowledge areas (REST APIs, testing, OpenAPI, LLMs)	Multiple choice	Background
Q1.3	Years of experience in software engineering	Single choice	Experience
Q1.4	Main programming languages	Multiple choice	Background
Q1.5	Works with integration tests	Yes/No	Experience
Q1.6	Has used OAS/OpenAPI for specification or API understanding	Yes/No	Experience
Q1.7	Has used LLMs for technical tasks	Yes/No	Experience
Q1.8	Has used TSL or similar test specification criteria	Yes/No	Experience

The inclusion of each profiling question was motivated by its relevance to interpreting the acceptance of RestTSLLM (RQ3):

- **Q1.1 (role)** — The RestTSLLM approach is relevant to the several roles involved in the REST API development lifecycle (e.g., developers, QA, test analyst, technical leads, researcher, teacher, or student). Acceptance may differ across these roles, so this question enabled role-based stratification.
- **Q1.3 (years of experience)** — Seniority may influence how practitioners perceive and trust a novel, AI-based approach; more experienced respondents may, for instance, have a more positive or negative perception given their broader and more detailed experience in the technical domains involved.

- **Q1.2 (knowledge areas)** — This question confirmed that respondents had sufficient context across the four pillars underlying the study — REST APIs, testing, OAS, and LLMs — to provide informed opinions.
- **Q1.4 (technical stack)** — This question characterized the technical diversity of the sample. Although the approach was demonstrated using C#/.NET, it is designed to be language-agnostic, so it was useful to know whether opinions originated from a diverse set of stacks.
- **Q1.5 (integration testing)** — Because the approach generates integration tests, direct experience with this activity is central to a meaningful evaluation of its usefulness and the trust placed in its outputs.
- **Q1.6 (OAS/OpenAPI)** — The OpenAPI Specification is the input artifact consumed by RestTSLLM. Familiarity with it is relevant to assess whether the input requirement is realistic and whether the approach is feasible in the respondent's context.
- **Q1.7 (LLMs)** — As the approach is LLM-based, prior exposure to LLMs for technical tasks may shape trust and adoption intention, and helps interpret eventual skepticism toward the generated tests.
- **Q1.8 (TSL)** — TSL is the intermediate representation that characterizes the pipeline's two-step design. Familiarity with TSL or similar test-specification criteria can impact the acceptance or understanding of the potential benefits of using the intermediate step.

The role options for Q1.1 included: Software Engineer; Test Engineer or QA; Tech Lead or Tech Manager; Software Architect, Staff, or Principal Engineer; DevOps Engineer or SRE; Product Owner/Manager; Teacher and/or Researcher in Software Engineering; Undergraduate or graduate student in Engineering or Computer Science; and Other (open text).

The experience ranges for Q1.3 were: less than 1 year; between 1 and 2 years; between 2 and 5 years; between 5 and 10 years; and more than 10 years.

6.3.2 Block 2 — Adoption and Usage Intention

Before answering Block 2, participants were presented with a concise description of the RestTSLLM approach, including: (i) a textual explanation of the two-step pipeline with

illustrations, (ii) an example OAS fragment, (iii) the corresponding TSL generated by the LLM, and (iv) the resulting executable integration test in C#. This contextualization ensured that responses reflected informed perceptions rather than general attitudes toward LLM-based testing. All instructions and descriptions can be viewed in the Appendix D.

Eliminatory Comprehension Check (Q2.0). Prior to the Likert-scale items, a single-choice comprehension check (Q2.0) asked participants to identify which of four options best summarizes the RestTSLLM approach. The three distractors described related but distinct strategies: (a) direct test generation without an intermediate step; (b) manual TSL specification with minimal LLM involvement; (c) LLM-based OAS enrichment for downstream tools. Only the fourth option — *"a strategy that uses LLMs to generate integration tests for REST APIs from OpenAPI specifications, using TSL as an intermediate representation to structure test scenarios before generating executable code"* — was correct. Responses with an incorrect answer were excluded from the analysis. This technique was adopted to validate that participants genuinely engaged with the description of the approach before expressing their opinions.

Likert-Scale Items (Q2.1–Q2.6). Six statements were evaluated on a five-point Likert scale, where 1 = *Strongly Disagree* and 5 = *Strongly Agree* (JOSHI et al., 2015). Table 6 presents the items.

Table 6: Block 2 — Likert-scale items on adoption and usage intention

ID	Statement
Q2.1	The RestTSLLM approach is useful for automating integration tests.
Q2.2	The RestTSLLM approach can be useful in my day-to-day work.
Q2.3	The separation between "designing test cases" and "generating test code" facilitates the use of LLMs for testing.
Q2.4	I would trust running and using tests automatically generated by the RestTSLLM approach.
Q2.5	I would use a tool implementing the RestTSLLM approach if it were available in my team.
Q2.6	I believe that integration test generation with RestTSLLM is applicable in industrial and large-scale scenarios.

Integration Modality Preference (Q2.7). A multiple-choice question (Q2.7) asked participants to select which integration modalities they would find most useful: IDE plugin;

agent or skill (via LLM extension); CLI; web service; CI/CD pipeline; or Other (open text).

6.3.3 Block 3 — Open-Ended Suggestions

Block 3 contained two optional open-ended questions to capture qualitative feedback:

- **Q3.1:** What aspects — positive or negative — would stand out most to you when using RestTSLLM to support test generation?
- **Q3.2:** What suggestions would you give to make the RestTSLLM approach more useful for your context?

6.4 Evaluating the Survey Instrument

The survey instrument underwent a two-stage evaluation before dissemination. First, the questionnaire was reviewed by a senior researcher with extensive experience in software engineering and empirical studies, who assessed the relevance, clarity, and completeness of each item, as well as its alignment with the research questions. This expert review supported the content validity of the instrument (KITCHENHAM; PFLEEGER, 2008), and the questions were refined according to the feedback received.

Second, a pilot study was conducted to validate the questionnaire's clarity, completeness, and estimated completion time. A small group of practitioners from different roles and experience levels (six practitioners) was invited to complete the pilot version and provide feedback on ambiguous wording, missing response options, and perceived difficulty of any item. Adjustments based on pilot feedback were applied before the final version was disseminated. Pilot responses were not included in the main analysis.

6.5 Obtaining Valid Data

6.5.1 Instrument and Platform

The questionnaire was implemented in Google Forms, which offered anonymous data collection (no email or personal information was stored), broad accessibility, and reliable response storage. The survey was entirely web-based and required no prior installation, ensuring minimal friction for participants.

6.5.2 Dissemination Strategy

The questionnaire was disseminated through two complementary channels: (i) *individual invitation* via LinkedIn direct messages to practitioners identified as matching the target profile; and (ii) *collective invitation* via technology-focused groups and communities on WhatsApp, Telegram, and Slack. Both channels used a standardized invitation text — available in the Appendix F — identifying the study's academic purpose, the approximate completion time (approximately 10 minutes), and the anonymity guarantee. The survey was open for responses from May 24 to June 6, 2026.

The invitation explicitly stated that the survey was appropriate for professionals with background in REST API development, integration test, OAS/Swagger/Postman tooling, or LLMs applied to development — matching the sampling criteria described in Section 6.2.1.

6.5.3 Comprehension Check

To ensure the validity of the collected data, responses that failed the eliminatory comprehension check (Q2.0, described in Section 6.3.2) were discarded, as they indicated that the participant had not genuinely engaged with the description of the approach. As all questions were mandatory, no partial or incomplete responses were collected.

6.6 Analysing the Data

The collected responses were exported from Google Forms into a spreadsheet and analyzed using Google Sheets, which was used to compute the descriptive statistics, perform the subgroup stratifications, and produce the charts presented in this study. The choice of a spreadsheet-based tool was motivated by the moderate volume of data and the predominantly descriptive nature of the analysis.

6.6.1 Quantitative Analysis

Responses to the Likert-scale items (Q2.1–Q2.6) were primarily analyzed through frequency distributions, reporting the proportion of respondents per agreement level. The median was used as the central tendency measure for each item, given the ordinal nature of Likert data (JOSHI et al., 2015). Means were reported as supplementary descriptors, following

common practice in software engineering surveys (KITCHENHAM; PFLEEGER, 2008).

To investigate whether acceptance levels differ across subgroups, responses were stratified by role (Q1.1), declared knowledge areas (Q1.2), and years of experience (Q1.3). Question Q1.4 (primary programming languages) was reported descriptively to characterize the sample's technical profile, but was not used as a stratification variable, given that the approach is designed to be language-agnostic. Profile questions Q1.5–Q1.8 (binary items on hands-on experience with integration testing, OAS usage, LLM usage, and TSL usage) were used to contrast acceptance among participants with and without direct experience in each relevant domain.

Results for Q2.7 (integration modality preference) were reported as frequency distributions showing the relative preference for each modality across the full sample, and stratified by role (Q1.1) to capture whether different professional profiles favor distinct integration approaches.

6.6.2 Qualitative Analysis

Open-ended responses from Q3.1 and Q3.2 were analyzed using open coding (GLASER; STRAUSS, 1967) to identify recurrent themes. Themes were derived inductively from the responses and grouped into positive perceptions, concerns, and suggested improvements. Representative excerpts are included in the results chapter to illustrate each theme.

6.7 Ethical Considerations

This survey was designed and conducted in compliance with the ethical requirements for studies with human participants. The study protocol was submitted for review and approved by the *Comitê de Ética e Pesquisa (CEP)* — from *Universidade Federal Fluminense (UFF)*, humans committee (Comitê de Humanas) — under registration number *CAAE 89588625.8.0000.8160*, in accordance with Brazilian research regulations. The key ethical safeguards were:

- **Anonymity:** The survey collected no personally identifiable information. No name, email address, or any other identifying data was recorded. Participation in the Google Forms survey was entirely anonymous.
- **Informed consent:** Participants were presented with a *Free and Informed Consent Form* (TCLE – Termo de Consentimento Livre e Esclarecido) – at the beginning

of the questionnaire. Submission of the form constituted implicit agreement to the consent terms. The *Free and Informed Consent Form* used can be found in the Appendix E.

- **Voluntary participation:** Participation was entirely voluntary. Participants could discontinue at any time and without explanation, and no partial data from withdrawn participants was retained.
- **Purpose limitation:** All collected data was used exclusively for academic research purposes related to this dissertation and potential scientific publications derived from it.
- **Risk minimization:** Participation involved no physical, psychological, or financial risk beyond the minimal effort of completing an online questionnaire (estimated at approximately 10 minutes of internet access).

7 Results and Discussion

This chapter presents and discusses the results obtained by applying the proposed RestTSLLM approach. The findings are organized according to the research questions (RQ) defined in this study.

To answer **RQ1** and **RQ2** we computed a final score (average calculated score) for each model, based on a weighted score of success rate, coverage, and mutation score. The aggregated results, ordered from highest to lowest, are presented in Table 7 and available in a chart in Figure 7. The detailed results by LLM and project are available in Appendix B and in a public PDF file on our GitHub repository[6]. Although it is not part of the evaluation criteria, we included the number of tests generated and the total cost per generation to support our analysis. For simplification purposes, in Table 7, an acronym was used to represent the average of each metric: $S \rightarrow$ calculated score; $SR \rightarrow$ success rate; $C \rightarrow$ coverage; $M \rightarrow$ mutation score; $T \rightarrow$ number of tests; and $TC \rightarrow$ total cost (USD) .

Table 7: Average of the metrics for the tests generated by LLMs

Model	**S**	SR	C	M	T	TC
Claude 3.5 Sonnet	70,9%	100%	71,7%	40,8%	38,3	$ 0,47
Deepseek R1	67,1%	97,0%	67,5%	36,9%	33,7	$ 0,78
Qwen 2.5 32b	65,8%	95,5%	68,7%	33,3%	34,7	$ 0,09
Sabiá 3	65,5%	97,5%	64,3%	34,7%	21,7	$ 0,08
LLaMA 3.2 90b	63,9%	97,5%	66,1%	28,0%	36,2	$ 0,02
GPT 4o	63,4%	98,0%	59,3%	32,9%	21,8	$ 0,25
Gemini 1.5 Pro	63,0%	96,5%	70,0%	22,6%	42,8	$ 0,19
Mistral Large	62,2%	98,0%	63,0%	25,5%	43,3	$ 0,31

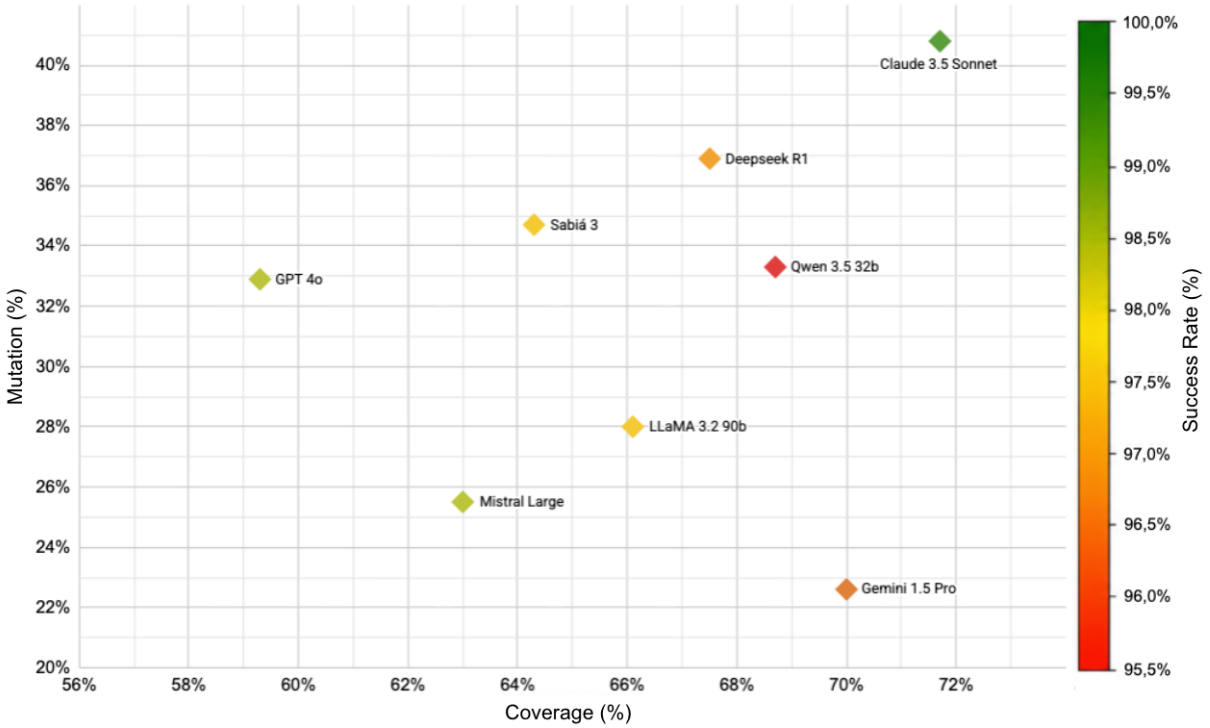

Figure 7: Scatter plot of average Coverage (%) versus average Mutation Score (%) for each of the eight evaluated LLMs

7.1 RQ1: Are LLMs effective in generating integration tests that reflect the intended business logic and context?

From a qualitative perspective, the application of *few-shot* and *decomposed prompting* techniques—using examples with expected outputs in a consistent and functional format—demonstrated satisfactory results. All LLMs managed to preserve the expected structure and generate compilable code for integration test execution.

The content generated was read and manually reviewed. The test cases were aligned with the business logic described in the OpenAPI Specifications, demonstrating that the models understood the intended behavior of the APIs. In the manual review, we validated the alignment between the generated content and the business rules by analyzing details such as authentication criteria, status codes, input and output properties, as well as whether both success scenarios and relevant edge cases made sense in the context of the specification. All results were satisfactory in terms of readability, clarity, and adherence to the test patterns presented in the prompts. Among the evaluated APIs, only *todo-api* had pre-existing tests, which showed no structural or naming convention similarity with the outputs generated by the models. All generated test cases and integration code are available on GitHub[5].

In addition to this subjective evaluation, the effectiveness of the tests generated by each model was measured using three key metrics: success rate, coverage, and mutation score. All models achieved average success rates above 95,5%, with balanced coverage and mutation scores in most cases. Only 2,38% of all generated tests had some kind of failure, which we detail in Subsection 7.2.1. This indicates that LLMs are capable of producing tests that not only execute successfully but also explore the SUT test meaningfully.

It is important to note that the tests had a relatively low mutation rate given the black-box test generation, which, by design, omits the implementation aspects of the test design, and may cause this side effect, indicating a possible gap between the specification and the implementation. Despite this, compared to the only project that had tests, *todo-api*, all LLMs achieved better results than the pre-existing tests (LLMs with a minimum score of 65.3% and original *todo-api* tests with score of 57.2%).

RQ1

Are LLMs effective in generating integration tests that reflect the intended business logic and context?

All LLMs generated compilable, executable integration tests, with success rates above 95,5%

Generated tests showed adequate structure, readability, and alignment with the business logic described in the OAS and example pattern

The results demonstrate that LLMs can be effective in generating integration tests, with average calculated scores ranging from 62,2% to 70,9%

7.2 RQ2: Which LLM is the most effective in generating integration tests?

To determine the most effective LLMs, we computed a final score (average calculated score) for each model, based on a weighted score of success rate, coverage, and mutation score. The results, ordered from highest to lowest, are presented in Table 7.

The top-performing models were Claude 3.5 Sonnet, Deepseek R1, Qwen 2.5 32b, and Sabiá 3, with calculated scores ranging from 65,5% to 70,9%. Among them, Claude 3.5

Sonnet stood out as the most effective, ranking first in all metrics and being the only model that produced no failed tests (Table 9). However, we observed that Deepseek R1, Qwen 2.5 32b, and Sabiá 3 achieved results close to Claude in all metrics, with a maximum difference of 7,5%. The ranking of models by individual metrics and their relative differences from the top performer is shown in Table 8.

Table 8: Models performance ranking by metric, and difference (Δ) from first position

Position	S	ΔS	SR	ΔSR	C	ΔC	M	ΔM
1°	Claude	–%	Claude	–%	Claude	–%	Claude	–%
2°	Deepseek	-3,8%	Mistral	-2,0%	Gemini	-1,7%	Deepseek	-3,9%
3°	Qwen	-5,1%	GPT	-2,0%	Qwen	-3,0%	Sabiá	-6,1%
4°	Sabiá	-5,4%	Sabiá	-2,5%	Deepseek	-4,2%	Qwen	-7,5%
5°	LLaMA	-7,0%	LLaMA	-2,5%	LLaMA	-5,6%	GPT	-7,9%
6°	GPT	-7,5%	Deepseek	-3,0%	Sabiá	-7,4%	LLaMA	-12,8%
7°	Gemini	-7,9%	Gemini	-3,5%	Mistral	-8,7%	Mistral	-15,3%
8°	Mistral	-8,7%	Qwen	-4,5%	GPT	-12,4%	Gemini	-18,2%

Although the LLMs Mistral Large, Gemini 1.5 Pro, GPT 4o, and LLaMA 3.2 90b presented lower average scores, they still achieved solid results, particularly in success rate and, in some cases, coverage or mutation score. For example, Gemini ranked second in coverage with only -1,7% below the best model. Similarly, Qwen and GPT showed competitive mutation scores with differences of only -7,5% and -7,9% respectively. Nevertheless, all models demonstrated satisfactory performance overall, with average calculated score values ranging between 62,2%, and 70,9%, a difference of at most 8,7%.

In addition to performance, we also tracked the average total cost of processing each project with each LLM. As shown in Table 7, the execution cost per project remained very low across all models. Even the highest, Deepseek, remained under one dollar ($0,78), while models like LLaMA, Sabiá, and Qwen stood out for delivering competitive results at extremely low costs—each below $0,09 per execution. This reinforces the feasibility of adopting LLMs for test generation even in budget-constrained environments.

RQ2

Which LLM is the most effective in generating integration tests?

Claude 3.5 Sonnet stood out as the most effective model, ranking first in all metrics with a calculated score of 70,9% and zero test failures

Top-performing models were Claude 3.5 Sonnet, Deepseek R1, Qwen 2.5 32b, and Sabiá 3, with scores ranging from 65,5% to 70,9%

All models demonstrated satisfactory performance overall, with calculated scores between 62,2% and 70,9% (a maximum difference of 8,7%). Despite satisfactory results, other models (Mistral Large, Gemini 1.5 Pro, GPT 4o, and LLaMA 3.2 90b) are less suitable for this activity

7.2.1 Analysis of Test Failures

Out of the 1.635 tests generated, 39 failed to execute correctly, representing 2,38%. Table 9 shows the number of failed tests per model and Appendix C details the failures by project and model. All models except Claude 3.5 Sonnet produced at least one faulty test. Upon manual inspection of the failures, we identified six recurring categories of errors, as shown below:

- **15 failures – Property Length**: Validation of boundary values outside allowed ranges.
- **7 failures – Misinterpretation**: Incorrect logic or misunderstanding of the API specification.
- **5 failures – Authentication**: Missing or incorrect use of authentication.
- **5 failures – Property Requirement**: Misuse of required or optional fields.
- **4 failures – Required Characters**: Missing required characters (uppercase, digit, etc.).
- **3 failures – JSON Deserialization**: Errors in parsing response content.

These failure patterns are consistent with findings reported in the broader LLM test generation literature: incorrect assertions and logical misinterpretations have been identified as a major source of test failures (YUAN et al., 2024). In our study, input value-range validation (the "Property Length"category) was the most frequent failure, indicating that correctly encoding boundary-related validation logic remains challenging for the evaluated LLMs.

These results reinforce the need for complementary techniques when using LLMs to improve test generation, to evolve from satisfactory results to exceptional results, whether in optimizing inputs, presented examples, prompt engineering or better models.

Table 9: Failed tests produced by the LLMs

Model	Total of Tests	Failed Tests	% Failed Tests
Claude 3.5 Sonnet	230	0	0,0%
Deepseek R1	202	4	2,0%
Qwen 2.5 32b	208	8	3,8%
Sabiá 3	130	4	3,1%
LLaMA 3.2 90b	217	7	3,2%
GPT 4o	131	2	1,5%
Gemini 1.5 Pro	257	9	3,5%
Mistral Large	260	5	1,9%

7.3 RQ3: Level of Acceptance

This section reports the results of the acceptance survey whose design was described in Chapter 6, addressing **RQ3** — *What is the level of acceptance of the RestTSLLM approach among practitioners involved in the REST API development lifecycle?*

The analysis follows the plan defined in Section 6.6: Likert-scale items are examined through frequency distributions and the median as the central-tendency measure for ordinal data (JOSHI et al., 2015), with means reported as supplementary descriptors; multiple-choice and profiling items are reported as frequency distributions; and the open-ended responses are examined through open coding (GLASER; STRAUSS, 1967). Of the 108 responses received, 12 were discarded for failing the eliminatory comprehension check (Q2.0), yielding **96 valid responses** analyzed below. Results are organized by the sub-questions RQ3.1–RQ3.5 defined in Section 6.1, followed by the qualitative analysis

and a synthesis that answers RQ3.

7.3.1 Participant Profile

This section characterizes the 96 valid respondents. The sample is dominated by software engineers (41), followed by QA/test engineers (14) and technical leads or managers (14), with the remaining responses distributed across architects/staff engineers (7), researchers/professors (8), students (7), and DevOps/SRE practitioners (5) as shown in Figure 8.

Figure 9 shows that the sample is notably senior: 66 respondents (68.8%) reported more than five years of experience in software engineering, and 43 (44.8%) reported more than ten years.

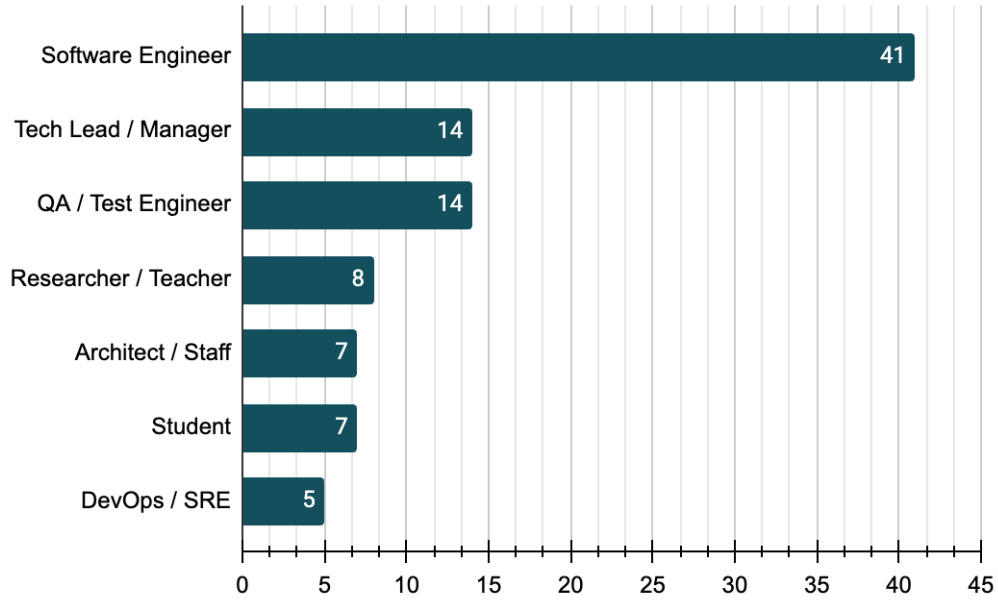

Figure 8: Participant profile — Current role

Regarding declared knowledge areas (Q1.2), as can be seen in Figure 10, 91 respondents (94.8%) reported knowledge of or interest in LLMs applied to development, 91 (94.8%) familiarity with OAS/Swagger/Postman tooling, 79 (82.3%) participation in the REST API development lifecycle, and 70 (72.9%) practical experience with integration testing or QA.

The binary experience items (Q1.5–Q1.8) confirmed this profile and showed that the main technical pillars used in the RestTSLLM approach are based on hands-on experience among participants, with a clear exception for the intermediate language (TSL), where only a minority worked directly. This reinforces the fact that the inclusion of this step

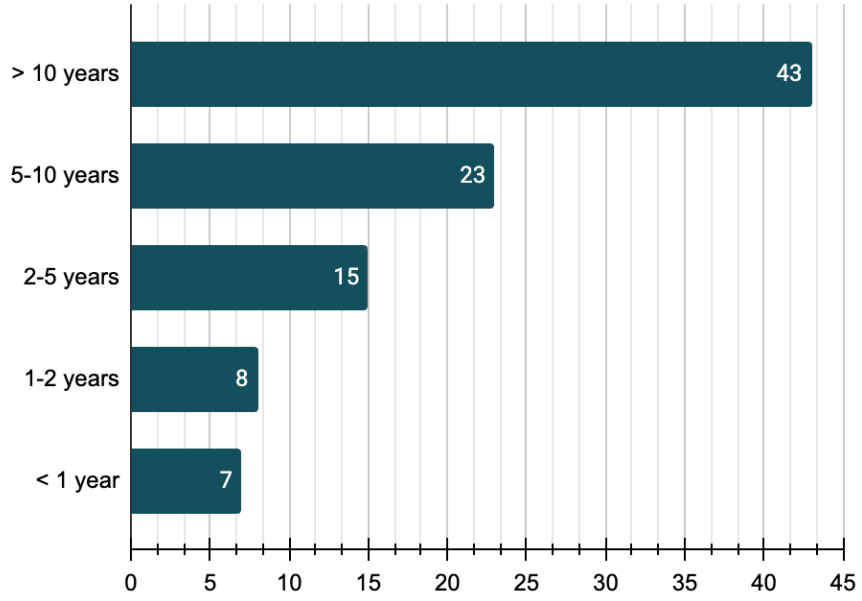

Figure 9: Participant profile — Years of experience in Software Engineering

is absent in real-world practice, and its strategic use with LLM can be a differentiating factor in the approach. Figure 11 describes the results obtained regarding the participants' hands-on experience: 89 (92.7%) had used LLMs for technical tasks, 83 (86.5%) had used OAS, and 66 respondents (68.8%) work directly with integration tests. In contrast, only 25 respondents (26.0%) had previously used TSL or a similar test-specification criterion, indicating that the intermediate representation central to the approach was unfamiliar to roughly three quarters of the sample — a relevant consideration when interpreting the perception of the pipeline design (RQ3.3).

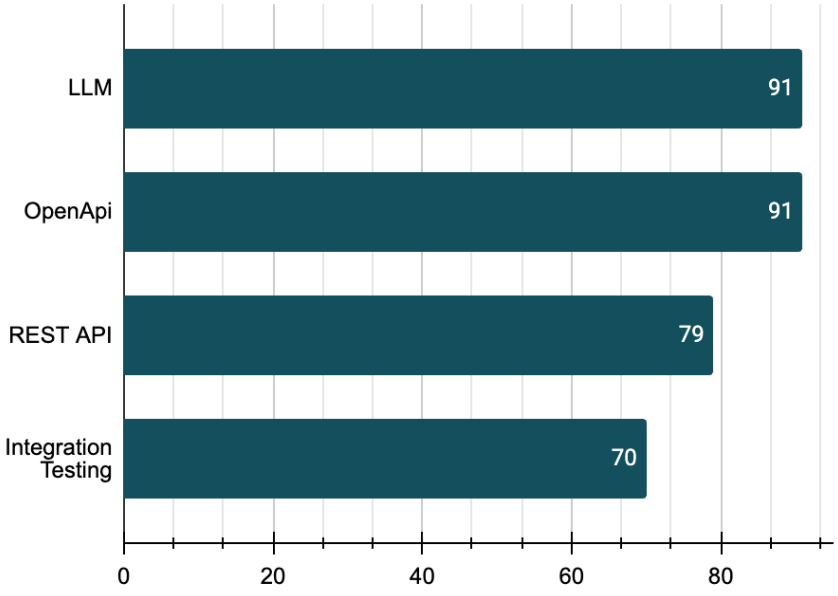

Figure 10: Participant profile — Declared Knowledge

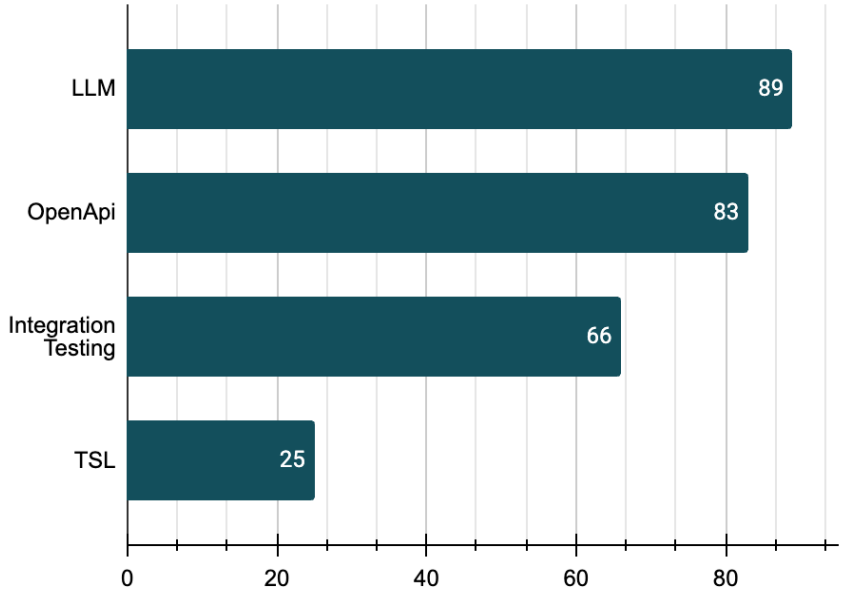

Figure 11: Participant profile — Hands-on experience / work directly

Regarding used programming languages, the predominant technology stacks were JavaScript (52), C# (41), Python (32), and Java (30), while the others were mostly cited together with one of these four most cited (97.9%, 94 responses), confirming that the opinions originate from a technically diverse audience rather than from a single ecosystem, as shown in Figure 12.

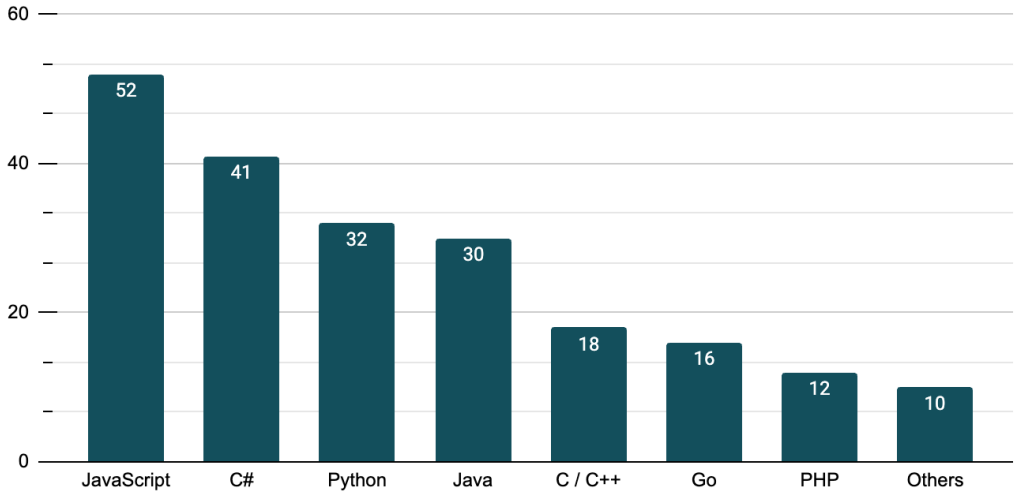

Figure 12: Participant profile — Stack / programming language

7.3.2 Overall Acceptance

Figure 13 presents the distribution of agreement for the six Likert items, and Table 10 summarizes their descriptive statistics. Across all items, the level of agreement was high:

every item obtained a median of 4 or 5, a mean between 3.92 and 4.50, and a proportion of agreement (responses of *Agree* or *Strongly Agree*) of at least 74%. The overall mean across the six items was **4.28** on the 1–5 scale, indicating a generally positive reception of the RestTSLLM approach among practitioners.

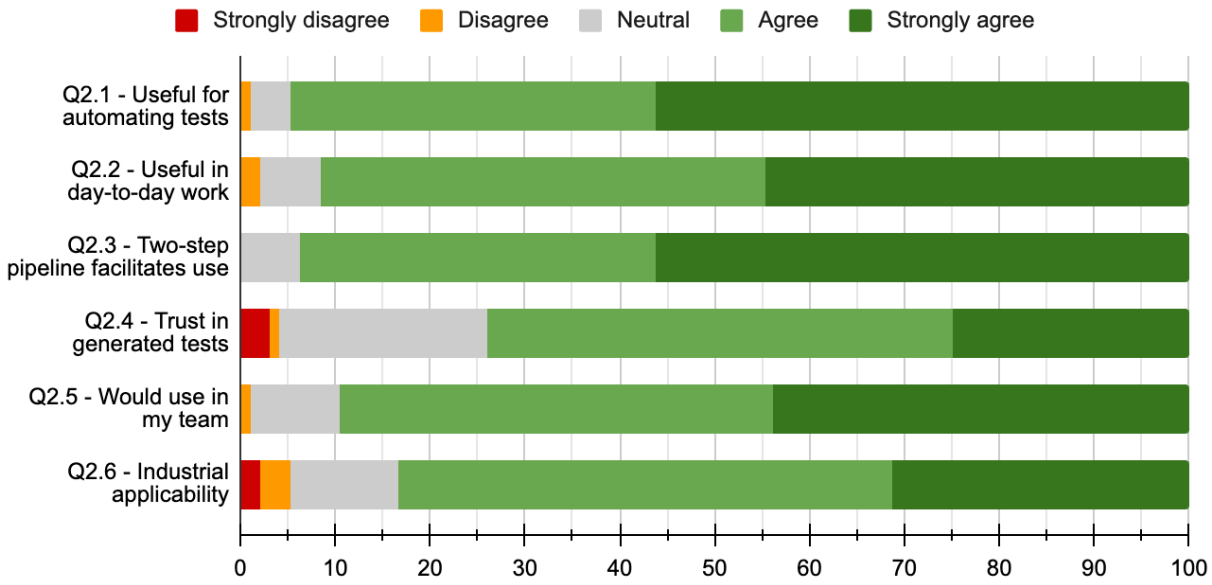

Figure 13: Distribution of responses to the Likert-scale items (Q2.1–Q2.6), n = 96.

Table 10: Descriptive statistics of the Likert-scale items (n = 96). Mdn = median; SD = standard deviation.

Item	Statement	Mdn	Mean	SD	% Disagree	% Agree
Q2.1	Useful for automating tests	5	4.50	0.63	1.0	**94.8**
Q2.2	Useful in day-to-day work	4	4.34	0.69	2.1	91.7
Q2.3	Two-step pipeline facilitates use	5	4.50	0.61	0.0	93.8
Q2.4	Trust in generated tests	4	3.92	0.89	4.2	74.0
Q2.5	Would use in my team	4	4.32	0.68	1.0	89.6
Q2.6	Industrial applicability	4	4.07	0.86	5.2	83.3
Overall (mean of items)		–	**4.28**	–	–	–

7.3.3 Perceived Utility (RQ3.1)

RQ3.1 investigates how practitioners perceive the utility of RestTSLLM and is addressed by items Q2.1 (*useful for automating tests*), Q2.2 (*useful in day-to-day work*), and Q2.6 (*industrial applicability*). The perception of utility was the strongest dimension of the survey. Item Q2.1 (*the approach is useful for automating integration tests*) obtained

the highest agreement of the entire instrument, with 94.8% agreement (median 5, mean 4.50) and a single respondent disagreeing. The perceived usefulness in the respondents' own daily work (Q2.2) was also high, at 91.7% agreement (median 4, mean 4.34). The most demanding utility item, Q2.6 (*applicable in industrial and large-scale scenarios*), obtained a lower but still strong agreement of 83.3% (median 4, mean 4.07), with 11.5% of neutral responses — consistent with the qualitative concerns about large and complex systems discussed in Section 7.3.8. Overall, practitioners clearly recognized the utility of the approach, with a slight and expected attenuation when projecting that utility onto large-scale industrial settings.

7.3.4 Trust and Intention to Use (RQ3.2)

RQ3.2 examines the extent to which practitioners trust and intend to use a tool implementing RestTSLLM, addressed by items Q2.4 (*trust in generated tests*) and Q2.5 (*would use in my team*). Intention to use was high: 89.6% of respondents agreed that they would use a tool implementing the approach if it were available to their team (Q2.5, median 4, mean 4.32). Trust in the generated tests (Q2.4), however, was the *lowest-rated item of the survey*, with 74.0% agreement, the highest proportion of neutral responses (21.9%), and the lowest mean (3.92) and highest dispersion (SD 0.89).

This gap between a high intention to use and a more reserved trust is one of the most informative findings of the survey. It indicates that practitioners are willing to adopt the approach but expect to retain a human-in-the-loop validation step before relying on the generated tests — a reading strongly corroborated by the open-ended responses (Section 7.3.8), where the need for human review was the most frequent concern. This finding is consistent with the broader literature on the adoption of AI-assisted software engineering tools and reinforces the relevance of the future work on automatic error correction and validation already outlined in Chapter 8.

7.3.5 Two-Step Pipeline Design (RQ3.3)

RQ3.3 asks whether practitioners consider the two-step pipeline (OAS → TSL → tests) a useful design decision, addressed by item Q2.3 (*two-step pipeline facilitates use*). Despite only 26% of respondents having prior experience with TSL (Section 7.3.1), the separation between *designing test cases* and *generating test code* was very well received: Q2.3 obtained 93.8% agreement (median 5, mean 4.50) and, notably, *not a single respondent disagreed*

with the statement. This makes Q2.3 the item with the most favorable disagreement profile of the instrument. The qualitative responses explain this result: many participants spontaneously identified the intermediate TSL layer as the central strength of the approach, citing its role in reducing hallucinations, enabling structural review before code generation, and decoupling reasoning from implementation. The design decision that most distinguishes RestTSLLM from a direct OAS-to-code prompt was therefore validated by the practitioners, even by those unfamiliar with the formalism.

7.3.6 Preferred Integration Modalities (RQ3.4)

RQ3.4 investigates which integration modalities practitioners would find most useful (Q2.7, multiple choice). Figure 14 presents the results. The most preferred modality was an *agent or skill* integrated into an LLM platform, selected by 74 respondents (77%), closely followed by an *IDE plugin* with 64 (67%). Integration into *CI/CD pipelines* (45 respondents; 47%) and a *CLI* (44 respondents; 46%) formed a second tier, while a standalone *web service* was the least preferred (15 respondents; 16%).

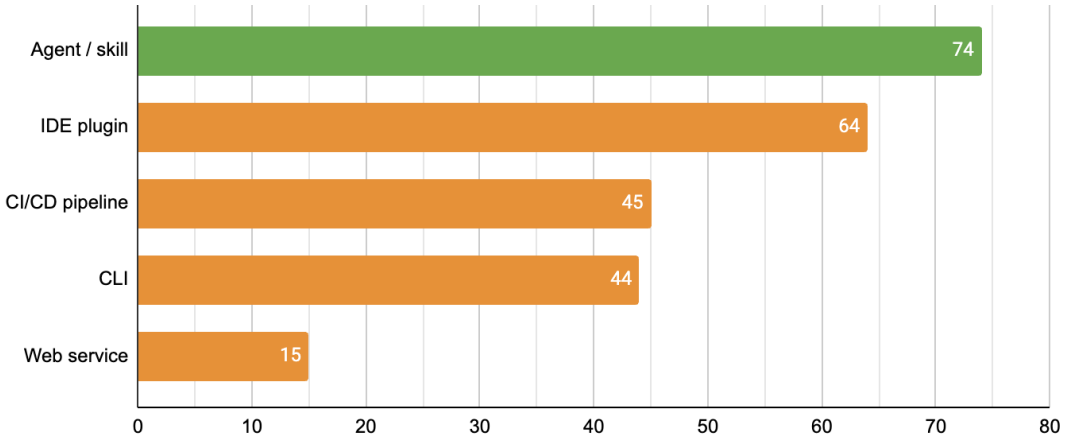

Figure 14: Preferred integration modalities (Q2.7), multiple choice, n = 96

The strong preference for agents/skills and IDE plugins indicates that practitioners envision RestTSLLM embedded directly into their development environment and interactive AI workflows, rather than as an isolated external service. This preference directly informs the prioritization of the future-work directions discussed in Chapter 8, particularly the development of skills, agents, and IDE plugins.

Acceptance Across Professional Profiles

7.3.7 Acceptance Across Professional Profiles (RQ3.5)

RQ3.5 investigates whether acceptance levels differ across professional profiles. We stratify the responses along the four profiling dimensions defined in Block 1: professional *role* (Q1.1), *years of experience* (Q1.3), *background breadth* (Q1.2), and *hands-on experience* with the underlying technologies (Q1.5–Q1.8). To avoid redundancy between the last two dimensions — which span related domains (testing, OAS, LLMs) — we deliberately treat them at different granularities: Q1.2 captures the *breadth* of a respondent's declared knowledge (*how many* of the four areas they are familiar with, i.e., awareness), whereas Q1.5–Q1.8 capture the *depth* of concrete prior use of *each specific* technology (i.e., practice). Because the subgroups are small and the sampling is non-probabilistic, these comparisons are interpreted as *exploratory* rather than conclusive (Section 7.4.2).

By role (Q1.1). Acceptance was positive across *all* roles, with overall means ranging from 4.05 to 4.70 (Figure 15). **QA/test engineers** reported the lowest overall acceptance (4.05) and, in particular, the lowest trust in the generated tests (mean Q2.4 of 3.43) — a coherent and arguably healthy result, since the professionals whose core responsibility is test quality are the most cautious about delegating it to an LLM. At the opposite end, DevOps/SRE practitioners (4.70) and students (4.43) were the most enthusiastic.

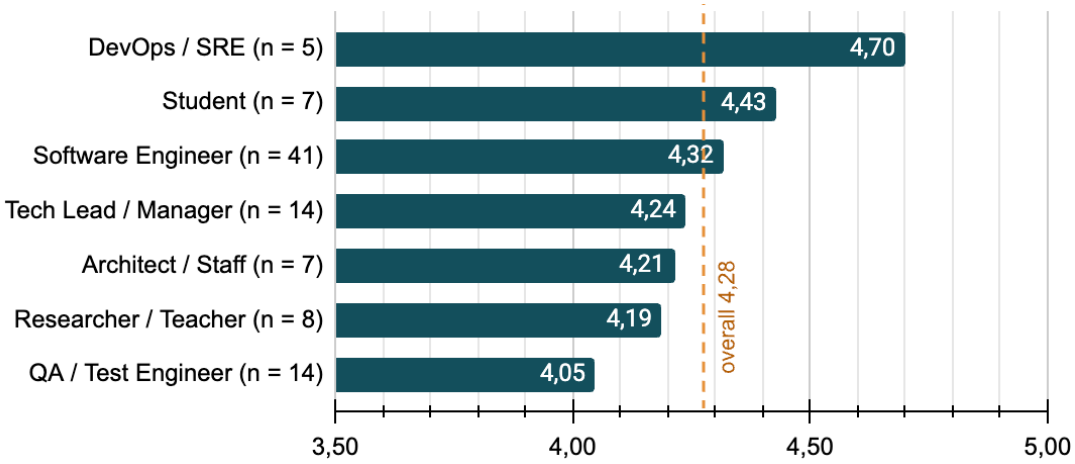

Figure 15: Overall acceptance (mean of Q2.1–Q2.6) by professional role. The dashed line marks the overall mean (4.28)

By years of experience (Q1.3). The relationship between seniority and acceptance was *not* monotonic (Figure 16). The most reserved group was the 5–10-year cohort (overall 4.16, trust 3.78), while early-career respondents were the most enthusiastic (1–2 years: overall 4.58, trust 4.38). The most senior cohort (>10 years) sat near the overall mean (4.25, trust 3.93). Even so, acceptance remained clearly positive across all experience

levels (every cohort mean ≥ 4.16). Given the small size of the least-experienced groups, these differences should be read with caution: at most, they suggest that enthusiasm is somewhat higher early in a career and slightly more tempered around the 5–10-year mark, rather than a clear effect of seniority.

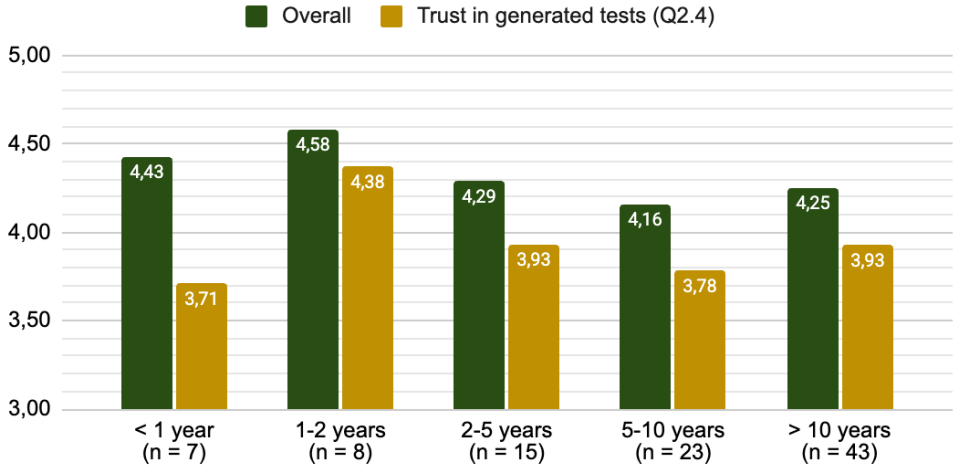

Figure 16: Overall acceptance and trust (Q2.4) by years of experience in software engineering

By hands-on experience — depth (Q1.5–Q1.8). This dimension produced the clearest signal, particularly on the trust item (Figure 17). Trust in the generated tests was substantially higher among respondents who had previously used LLMs (3.97 versus 3.29 for those who had not) and OAS (3.99 versus 3.46), whereas prior hands-on experience with integration testing (3.94 versus 3.87) and TSL (3.88 versus 3.93) made essentially no difference. Trust is thus shaped less by general testing practice and more by concrete familiarity with the two technologies the approach is built upon — LLMs and OAS.

By background breadth — awareness (Q1.2). Stratifying by the number of declared knowledge areas (out of the four in Q1.2) revealed no clear acceptance gradient (Figure 18): respondents declaring one or two areas (n = 15) averaged 4.44, those declaring three (n = 20) averaged 4.17, and those declaring all four (n = 61) averaged 4.27. Acceptance was therefore high and stable regardless of how broad a respondent's background was. The only area whose mere *declared* familiarity coincided with higher acceptance was LLM/AI knowledge (4.30 with versus 3.83 without), echoing — at the level of awareness — the same effect that hands-on LLM experience produced on trust in the previous dimension.

Summary. The four dimensions tell a consistent story: acceptance of RestTSLLM is high and broadly distributed — it does not depend substantially on a respondent's role, seniority, or breadth of background. What moves the more demanding *trust* dimension is

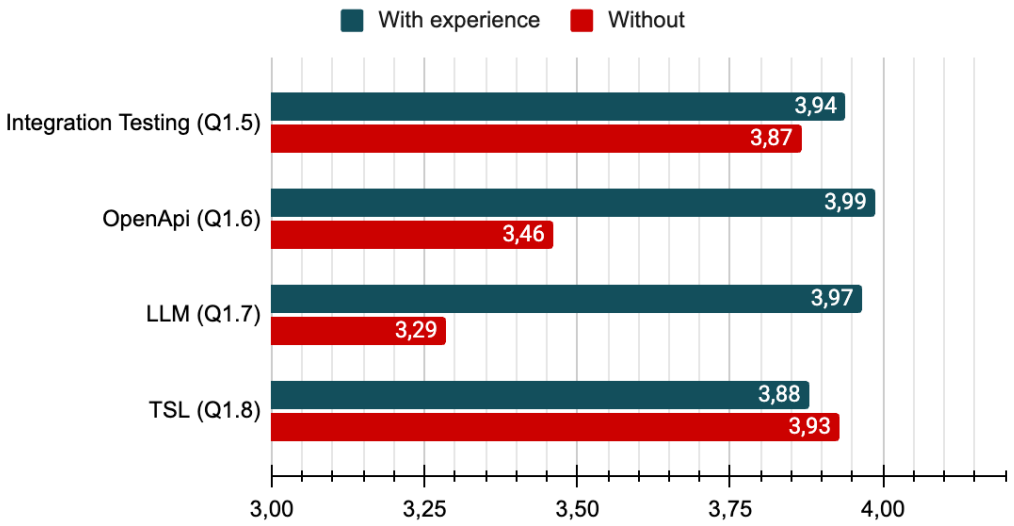

Figure 17: Trust in the generated tests (Q2.4) by prior hands-on experience with each underlying technology

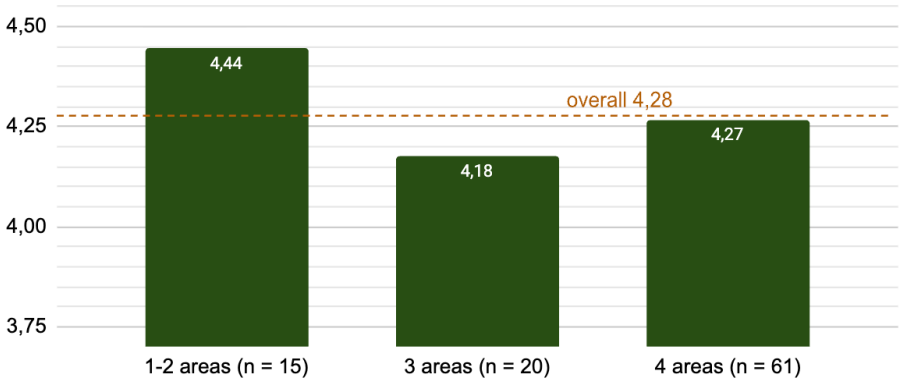

Figure 18: Overall acceptance (mean of Q2.1–Q2.6) by background breadth (number of knowledge areas declared in Q1.2). The dashed line marks the overall mean (4.28)

concrete, hands-on familiarity with the technologies underlying the approach, especially LLMs and OAS. This carries a practical implication for adoption: trust is likely to grow as practitioners gain direct exposure to these technologies, which should inform how the approach is introduced to a team.

7.3.8 Qualitative Analysis (Q3.1 and Q3.2)

While the Likert items quantify *how much* practitioners accept the RestTSLLM approach, the two open-ended questions reveal *why*. Q3.1 asked what would stand out — positively or negatively — when using the approach, and Q3.2 asked for suggestions to make it more useful. The responses were analyzed through inductive open coding (GLASER; STRAUSS, 1967; SALDAÑA, 2013): each response was read in its original language,

broken into meaning units, and assigned codes that were progressively grouped into sub-categories and three overarching categories — *positive perceptions*, *concerns*, and *suggestions*, as shown in Figure 19. Drawing on the principles of thematic analysis in software engineering (CRUZES; DYBÅ, 2011), and in line with our exploratory aim, the analysis foregrounds the *meaning* of the themes rather than their frequency; counts are used only sparingly, as a secondary indication of how widely a theme was shared — an approach consistent with prior open-coding studies of practitioner surveys in software engineering (TRINKENREICH et al., 2022). The complete coding — every response, the excerpt that motivated each code, and the resulting category and sub-category — is provided in Appendix G; representative excerpts are quoted below (respondents are identified by an anonymous code, e.g., R038, and quotes were translated from Portuguese by the author).

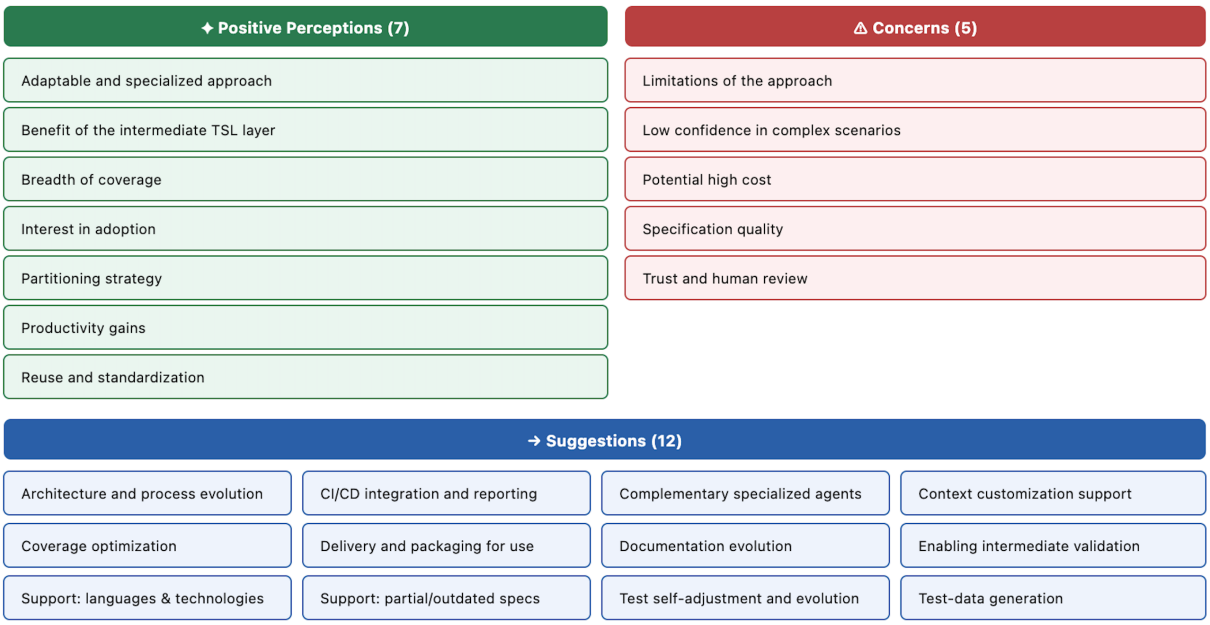

Figure 19: Categories identified by inductive open coding of the open-ended survey responses: positive perceptions (7 categories, Q3.1), concerns (5 categories, Q3.1), and suggestions for improvement (12 categories, Q3.2).

Positive perceptions. The theme that practitioners articulated most vividly was the value of the intermediate TSL layer — the design decision that distinguishes RestTSLLM from a direct OAS-to-code prompt. Respondents described it not as a mere implementation detail but as the mechanism that makes LLM-generated tests trustworthy and inspectable. One participant captured the core argument precisely: *"the strongest point is the use of the TSL as an intermediate layer; forcing the LLM to structure the test logic before generating the executable code drastically reduces the risk of hallucinations"* (R038). Others emphasized the *auditability* this enables — the possibility of inspecting and validating

the scenarios before any code exists: *"this architectural decoupling allows the auditing and structural validation of the scenarios before the computational expense of generating the executable code"* (R095), and *"it makes the process more transparent, facilitates the review of the generated scenarios, and reduces the direct dependency on the LLM to produce correct code on the first try"* (R096). For some, the same separation was valued because it brings *determinism* to an otherwise stochastic process (*"isolation of the LLM and determinism through the TSL"*, R051) and because it specializes a general-purpose model for the task (R049). Notably, this appreciation was expressed even by participants who had never used TSL before, suggesting that the rationale for the intermediate step is intuitive once seen.

A second, broadly shared positive theme was the gain in *productivity*. Practitioners framed test creation as a costly, often-skipped activity, and saw the approach as a way to reclaim that effort: *"automating test generation with quality and safety is always a huge performance gain for the development cycle"* (R024), and, at scale, *"the manual effort to maintain test coverage becomes unsustainable [...] the proposed solution mitigates this problem by automating the cycle end to end"* (R094). Closely related was the perceived *breadth of coverage*: several respondents valued that the generated suites go beyond the obvious cases — *"scenarios that a programmer alone might not be able to identify"* (R044), with *"good variation, going beyond the most basic success and failure tests"* (R030) and the ability to *"interpret corner cases"* (R040). Participants repeatedly praised, as well, the fact that the approach *reuses an artifact they already maintain* — the OpenAPI Specification — turning existing documentation into tests and standardizing them across services (R001, R031, R047). Finally, some respondents valued the approach's *adaptability* and the pragmatism of its design: that it offers *"a very fluid and familiar direction to adapt to"* (R028), and that breaking the work down by endpoint groups is a sensible engineering choice — *"dividing large systems into smaller parts"* (R069) — because *"partitioning the endpoints also shows pragmatism"* in the face of the models' context-window limits (R095).

Concerns. The dominant concern was not about the quality of any individual output but about *trust and the role of the human*. Even enthusiastic respondents expected to keep a person in the loop before relying on the generated tests: *"I still wonder to what extent it would be possible to trust the generated tests without additional validation"* (R069), explicitly framing the issue as one of *human-in-the-loop* review, while others asked for empirical reassurance before adopting it — *"I would need evidence of how much it gets right and wrong in controlled scenarios"* (R076). This theme is the qualitative counterpart of the lower trust score observed quantitatively (Q2.4), and the two sources of evidence reinforce each other.

A second concern targeted the approach's *dependency on the input specification.* Because the pipeline begins from the OAS, its output is only as good as that contract: *"the approach becomes very dependent on the quality of the initial OpenAPI specification; if the Swagger/OpenAPI is outdated or poorly documented, the tool will generate incomplete or wrong tests"* (R038), and the quality of the result *"may vary depending on the level of detail of the specification"* (R043) — a fragility aggravated by the fact that specifications *"can quickly become outdated"* (R092). A third concern, raised by practitioners working in more demanding environments, was the approach's reach into *complex scenarios* that the evaluated projects did not exercise: asynchronous and messaging-based systems (*"how it handles tests [...] in code that works with microservices and messaging"*, R091), the need to provision mocks and preconditions (R060), and business flows that span endpoints placed in different partitions (R095).

Finally, respondents pointed to broader *limitations of the approach* and to its *cost.* On the former, they noted the non-deterministic nature of LLMs as fundamentally at odds with testing (*"the LLM is not deterministic and tests need to be deterministic"*, R059), together with a possible *learning curve* to operate the model effectively and the risk of *technology lock-in* from *"tying oneself to restricted and enterprise technologies"* (R026). On cost, a few participants warned that the multiple inference steps could become expensive at scale — an *"exorbitant cost"* for large systems (R052) and a *"potential increase in cost and latency due to the multiple inference steps"* (R095).

Suggestions. The suggestions were strikingly constructive and, in several cases, converged on the same insight that underpinned the positive perceptions: keep the human in control of the intermediate representation. The most recurrent suggestion was an explicit *review checkpoint over the generated TSL before* the executable code is produced — both to retain oversight and to avoid spending inference on a flawed specification: *"a simple checkpoint that allowed the developer to review and adjust the TSL generated in Step 2 before the LLM spends tokens"* (R038), echoed by requests to *"allow the user to review and edit the TSL cases before generating the code"* (R096) and to add *"a mechanism for validating the generated TSL before the conversion into code"* (R065). A second cluster asked for *integration into the development workflow*: CI/CD pipelines with coverage reporting and dashboards (R083, R059), including specific tooling such as Allure (R065) and contract validation in CI (R092).

Practitioners also proposed mechanisms to *reduce the residual manual effort* that fuels the trust concern: a *self-healing* loop that feeds a failing test back to the model

for automatic correction (*"a closed-loop feedback mechanism (self-healing)"*, R089; *"self-correction of failing tests"*, R080; R064), an *evaluator agent* that screens the generated cases before human review (*"the integration of evaluator agents (Agent-as-a-Judge)"*, R069; R020), and support to keep the suite current — generating tests incrementally for only the code changed in a commit (R094) and accommodating the *evolution of the APIs* over time (R084). Others focused on *packaging* the approach for everyday use — as an IDE plugin, a CLI, an LLM skill, a parameterizable generic prompt, or a self-contained module (R030, R025, R080, R051), or made freely available within existing AI platforms (R014) — and on *adapting* it to real contexts through configuration such as authentication, test data, environment, and naming conventions (R096, R082, R084) and support for partial or outdated specifications (R065, R083). A smaller set pointed to broader technological directions: support for additional languages beyond .NET (R043), for asynchronous inputs (R026), and alignment with engineering practices such as vertical slicing and test-driven development (R095).

Beyond reducing effort, respondents proposed refinements to the *coverage strategy* itself: prioritizing critical over trivial scenarios (*"classification by priority, tests for critical and non-critical scenarios"*, R003; *"or the most critical cases"*, R050), exposing a configurable target-coverage level (R050), and even measuring *business* coverage rather than mere code coverage — *"the business coverage, not the code coverage"* (R006). A recurrent practical need was first-class support for *test data*, since black-box generation alone does not provision the environment: an integrated synthetic-data generator (*"a tool that does seeding"*, R016; dynamic *"fakers"*, R089) and assistance with *"a pre-setup of the database or of external integrations"* (R026). Others addressed *documentation and accessibility* — documenting the generated tests in a readable format such as Gherkin (R020), improving the repository documentation (R041), and letting scenarios be expressed in natural language so that non-developers can take part (R086). Finally, in addition to the alignment with vertical slicing and test-driven development noted above, a few respondents proposed a *white-box pre-analysis* of the flow to ground the black-box tests (R026) and the extension of the approach to further domains, such as data-engineering pipelines (R070).

It is worth highlighting that these practitioner-driven suggestions align with the future-work directions independently outlined in Chapter 8 — reprompting for error correction, IDE plugins, skills and agents, support for incomplete specifications, and the combination and maintenance of generated suites over time — which lends external validation to the planned evolution of the approach.

7.3.9 Synthesis: Answering RQ3

Bringing the quantitative and qualitative evidence together provides a clear and nuanced answer to **RQ3** — *What is the level of acceptance of the RestTSLLM approach among practitioners involved in the REST API development lifecycle?* The RestTSLLM approach was well accepted by the surveyed practitioners: agreement was high and consistent across all evaluated dimensions, and the qualitative responses explain what drives that acceptance. Practitioners value the approach primarily because it *reuses an artifact they already maintain* to recover an expensive, frequently neglected activity, and because its *intermediate TSL representation* makes the use of LLMs for testing structured, inspectable, and less prone to hallucination. The design decision that most distinguishes the approach from a naive prompt is therefore also the one practitioners celebrated most — a convergence between the contribution of this work and the needs expressed by its intended users.

The analysis also surfaces a single, coherent reservation that runs through both data sources: a *trust gap*. Quantitatively, trust in the generated tests (Q2.4) was the lowest-rated item even as the intention to adopt the approach remained high; qualitatively, the need for human review was the most prominent concern. Importantly, this gap is not a rejection of the approach but a well-articulated condition for its adoption: practitioners are willing to use RestTSLLM, provided they retain a human-in-the-loop checkpoint — ideally over the TSL, before code is generated — and, in the longer term, automated support such as self-healing loops and evaluator agents. This reading is reinforced by the exploratory profile analysis (Section 7.3.7), which showed that trust grows with hands-on familiarity with the underlying technologies (LLMs and OAS), suggesting that the gap narrows as exposure increases rather than reflecting a fundamental objection.

In summary, the answer to RQ3 is that RestTSLLM achieves a high level of acceptance among professionals involved in the REST API development lifecycle, anchored in its perceived usefulness and in the intermediate representation that defines it, and conditioned on human oversight of the generated artifacts. The practitioners' suggestions do not merely confirm this acceptance — they chart a concrete, demand-driven roadmap (review checkpoints, workflow integration, automated validation) that coincides with the future work proposed in this dissertation, strengthening the case that the approach addresses a real and recognized need in practice.

RQ3

What is the level of acceptance of the RestTSLLM approach among practitioners involved in the REST API development lifecycle?

RestTSLLM was well accepted by the 96 surveyed practitioners, with agreement above 83% across all evaluated dimensions about perceived utility

Useful for automating tests (94.8% agreement) and the two-step TSL-based pipeline design (93.8% agreement, zero disagreements) were the strongest perceived strengths

Adoption is conditioned on human review of the generated artifacts, reflected in trust being the lowest-rated dimension (74.0% agreement)

7.4 Threats to Validity and Limitations

7.4.1 Threats to the Experimental Evaluation

This experiment presents promising results in the use of LLMs for integration test generation, but some limitations, and threats to the validity of the study must be considered.

LLM Selection: The choice of LLMs was primarily based on model popularity. However, this may affect generalizability of our results, as emerging, lesser-known models, or those specifically designed for coding or trained for testing tasks, may yield different outcomes. We acknowledge this as a limitation and suggest that future studies include such models to broaden the analysis.

LLM Version Staleness: The model versions evaluated in this study reflect those publicly available during the first half of 2025, when the experiment was conducted. Given the rapid pace of LLM development, newer versions of the same models — as well as entirely new models — may have been released since then, potentially offering improved performance on test generation tasks. Model versioning has been identified as a persistent reproducibility concern in the broader LLM-for-SE research community (SIDDIQ et al., 2025). The gap between data collection and publication was partly extended by the institutional ethics review process required for the acceptance survey. We acknowledge this as a temporal limitation; re-evaluating the approach with updated model versions is a

direction for future work.

Project Context: The study focused on six open-source projects with relatively simple REST APIs, selected based on public availability and objective criteria. While suitable for evaluating the proposed method, more complex industrial systems—with larger or asynchronous APIs, stricter security constraints, or broader business logic—may present challenges not covered here. We note this as a limitation in the applicability to enterprise-scale scenarios.

Technology Stack Dependency: The evaluation focused on .NET projects and xUnit tests in C#, which may limit generalization to other stacks. However, RestTSLLM is not inherently tied to this ecosystem, since its target language and framework can be changed by replacing the prompt examples and adapting the execution pipeline.

OpenAPI Specification Dependency: Our approach depends on the availability and quality of OpenAPI Specifications to drive test generation. Although OpenAPI Specification is widely adopted, some legacy systems may lack such documentation or provide outdated or incomplete specifications, which could hinder the effectiveness of the approach. Future adaptations may consider alternative specification formats or test extraction from codebases directly.

Prompt Engineering Dependency: The quality of the results is strongly influenced by prompt design. Variations in prompt wording, structure, and examples can influence the output quality, as can the language used—Portuguese, in our case. Although prompt design was carefully iterated and validated across models, we recognize that this process introduces variability and may require adjustments in other languages or technologies.

Result Randomness: The use of a temperature value of 1 during LLM execution introduces stochasticity, meaning that repeated executions with the same prompts may produce different outputs. This affects reproducibility and consistency. We partially mitigated this by using a fixed seed where supported and by repeating executions during validation. Nonetheless, some models exhibited variations across runs and experiments with these variations are important to expand the knowledge and impact of this parameter.

Limited Metric Evaluation: While success rate, branch coverage, and mutation score are well-established metrics in software testing, they do not fully capture other aspects such as readability, maintainability, runtime performance, or test execution cost. These complementary metrics and aspects could be explored in future work to broaden the quality analysis of the generated tests.

Black-box Generation and Mutation Score Ceiling: Because RestTSLLM generates tests from the OpenAPI Specification alone, adopting a black-box perspective, it has no visibility into the internal implementation of the SUT. As a consequence, mutants affecting logic that is not observable through the API contract—such as internal branches not reflected in responses or status codes—tend to survive, imposing an inherent upper bound on the achievable mutation score. The comparatively low mutation scores observed in some projects (Section 7.1) are therefore partly explained by this characteristic of black-box generation, rather than solely by weaknesses in the generated tests. Combining the approach with white-box information or source-code analysis is a possible way to mitigate this effect.

Subjectivity in Qualitative Analysis: The qualitative assessment of test cases—particularly for RQ1—was performed manually by the first author. Despite their 10+ years of experience in software engineering and REST API testing, the evaluation is inherently subjective and may vary if performed by other reviewers. To reduce bias, all results were thoroughly cross-checked with the corresponding OpenAPI Specifications and expected patterns.

Generalization of Results: The results reflect the behavior of the evaluated models within the context of the selected projects. While diverse, these projects may not represent all types of REST APIs or organizational contexts. As such, the findings may not generalize to environments with significantly different technical stacks or operational constraints.

Tokenization Limit: Models with limited context windows occasionally produced incomplete outputs due to prompt size. To address this, we segmented the generation by endpoint group, which improved consistency. However, larger projects may still require new strategies to manage prompt length effectively.

7.4.2 Threats to the Acceptance Survey

In addition to the threats related to the experimental evaluation, the acceptance survey (Chapter 6) is subject to its own validity concerns, discussed below. These threats stem primarily from the survey design and its inherent characteristics as an opinion-based study, and were considered following the recommendations of the ACM SIGSOFT Empirical Standards for Questionnaire Surveys (RALPH et al., 2020; ACM SIGSOFT EMPIRICAL STANDARDS, 2020).

Sampling and Self-Selection Bias: The survey adopted a non-probabilistic pur-

posive sampling strategy, with participants recruited through LinkedIn and technology-focused communities. As respondents voluntarily chose to participate, the sample may over-represent professionals already interested in LLMs, testing automation, or the topic itself, potentially biasing the results toward more favorable perceptions. Because the sample is not probabilistic, the findings cannot be statistically generalized to the entire population of REST API practitioners. To mitigate this, we disseminated the invitation across diverse channels and roles, defined explicit eligibility criteria, and report results disaggregated by participant profile (Section 6.6.1).

Sample Size and Representativeness: The survey collected 108 responses, of which 12 were discarded for failing the eliminatory comprehension check (Q2.0), resulting in 96 valid responses. Although this sample size is reasonable for an opinion survey in software engineering, it still limits the strength of the subgroup analyses (RQ3.5), since some professional profiles may be under-represented or yield few responses per stratum. We therefore interpret subgroup comparisons as exploratory rather than conclusive, and emphasize overall response diversity over per-group statistical power.

Social Desirability and Voluntary Response Bias: Respondents may tend to provide more favorable answers, particularly in an evaluative study about a proposed approach. Likewise, individuals with strong opinions—whether positive or negative—are often more likely to respond to voluntary surveys. The fully anonymous format, with no collection of personal or identifying data, was adopted specifically to reduce social desirability pressure and encourage honest responses.

Construct Validity of the Instrument: Acceptance is a latent construct measured indirectly through Likert-scale items derived from the survey goal. Some sub-questions are assessed by a single item (e.g., RQ3.3 by Q2.3), which may not fully capture the underlying construct, and no previously validated acceptance scale (such as the Technology Acceptance Model (DAVIS, 1989)) was formally instantiated. To mitigate this, the items were grounded in a concrete description of the approach, reviewed by a senior researcher for content validity, and refined through a pilot study (Section 6.4).

Researcher Bias in Instrument Design: As the questionnaire was designed by the proponent of the RestTSLLM approach, question wording could unintentionally steer respondents toward favorable answers. This threat was mitigated through neutral phrasing, an independent expert review by a senior researcher, and adjustments based on pilot feedback before dissemination.

Comprehension and Engagement: The eliminatory comprehension check (Q2.0)

was included to discard responses from participants who did not genuinely engage with the description of the approach. However, this filter may introduce false negatives (discarding attentive respondents who simply misread the question) or, less likely, false positives (respondents who guessed the correct option). The question was designed with a single unambiguously correct option and plausible distractors to minimize these effects.

Perception Based on Description Rather Than Use: Participants evaluated the approach based on a textual description, illustrative examples, and the pipeline rationale, rather than on hands-on use of a tool. Stated perceptions and intention to use may therefore differ from behavior in real adoption scenarios. We mitigated this by providing a detailed and concrete contextualization (including example OAS, TSL, and generated tests), but we acknowledge that intention to use is not equivalent to actual adoption, which remains a direction for future longitudinal investigation.

Analysis of Ordinal Data: Likert-scale responses are ordinal, and treating them as interval data (e.g., computing means) may violate the mathematical assumptions of such scales (KITCHENHAM; PFLEEGER, 2008). To address this, the analysis relies primarily on frequency distributions and the median as the central tendency measure, reporting means only as supplementary descriptors.

Single-Coder Qualitative Analysis: The open coding of the open-ended responses (Q3.1 and Q3.2) was performed by a single researcher. This contrasts with practices that employ multiple independent coders and a consensus or inter-rater agreement step — as in comparable open-coding studies of practitioner surveys in software engineering (TRINKENREICH et al., 2022) — and introduces a risk of subjectivity in how excerpts were interpreted and grouped into codes, sub-categories, and categories (SALDAÑA, 2013). To mitigate this threat and support auditability, the complete coding trail is made available: every response, the specific excerpt that motivated each code, and its resulting category and sub-category are reported in Appendix G and in the supplementary material. In addition, the analysis foregrounds the meaning of the themes over their frequency, and conclusions are illustrated with verbatim quotes so that readers can assess the link between the data and the derived interpretations.

7.5 Extension Work

The limitations identified in Section 7.4 motivated two follow-up studies conducted by undergraduate students at the Universidade Federal Fluminense, both supervised by the

same research group. Each study targets a distinct limitation of the original RestTSLLM approach and independently confirms or extends its findings in new directions.

7.5.1 Extending RestTSLLM to Java and Python

A first limitation of this dissertation concerns the **Technology Stack Dependency**: the approach was designed and evaluated exclusively for .NET and xUnit, which restricts its out-of-the-box applicability to teams using other languages and testing frameworks. Felix (2025) investigated whether the RestTSLLM approach could be adapted to generate integration tests in **Java** (using REST Assured and JUnit) and **Python** (using PyTest), while preserving the same three-stage pipeline of the original approach.

The adaptation consisted primarily of replacing the technology-specific examples in *Prompt 2* (Stage II) —responsible for converting TSL into executable test code—with analogous examples written in the target languages. The OAS-to-TSL step in *Prompt 1* (Stage II) was reused without modification, confirming that the intermediate TSL representation is language-independent. Two open-source REST API projects were selected for evaluation: the Swagger PetStore (Java) and the Bookstore API (Python).

The study used GPT-4o and GPT-5 as the generation models. A key finding was that the newer GPT-5 produced significantly higher initial correctness (57–58%) compared to GPT-4o (16%), with the former requiring substantially fewer manual corrections to reach a fully functional test suite. This result aligns with the **LLM Selection** threat identified in Section 7.4, reinforcing the importance of model capability on output quality.

After the necessary manual corrections, the obtained quality metrics were: for the Java project (Swagger PetStore), 58% line coverage, 44% mutation coverage, and 78% test strength; for the Python project (Bookstore API), 97% line coverage. The Java results are closely comparable to those obtained by GPT in the original study (average coverage of 59,3%), confirming that the adaptation preserves the effectiveness of the original pipeline. The considerably higher coverage obtained in Python is primarily explained by the smaller size and structural simplicity of the Bookstore API, as discussed by the author.

The study also identifies an additional nuance related to the **OpenAPI Specification Dependency** limitation: when the target OAS is very large (exceeding approximately 1,300 lines, as in the PetStore specification), the LLM tends to process only a subset of the specification within a single prompt execution, resulting in incomplete TSL coverage and consequently lower test coverage. This behavior reinforces the segmentation strategy

by endpoint group already adopted in this dissertation, and suggests that future work should explore using the complete source code as a complementary input—a direction also noted in the original work's future work.

The full implementation, adapted prompts, and experimental artifacts are publicly available on GitHub[1], providing a reusable starting point for teams wishing to apply RestTSLLM outside the .NET ecosystem (FELIX, 2025).

7.5.2 Extending RestTSLLM to Asynchronous and Event-Driven APIs

A second limitation of this dissertation is the **Project Context** constraint: the evaluated REST APIs were relatively simple, synchronous, and stateless, which may not reflect the complexity of enterprise-scale systems. In particular, modern distributed architectures increasingly rely on asynchronous communication patterns—such as message queues (e.g., RabbitMQ, Kafka) and event-driven APIs—that are not representable through the synchronous request-response model assumed by this study's prompts and test structure.

In parallel, an ongoing undergraduate thesis Calvet (2026) addresses this gap by extending the RestTSLLM approach to support REST APIs with asynchronous behavior and messaging-based interactions. The work adapts the pipeline to handle systems in which an HTTP request triggers a background process communicated through a message broker, requiring tests to observe asynchronous outcomes rather than immediate synchronous responses. This extension broadens the practical scope of RestTSLLM toward production-grade, event-driven architectures that are common in microservice environments (NEWMAN, 2015).

Together, these two follow-up studies validate the extensibility premise of the RestTSLLM approach—that its three-stage pipeline is adaptable to new languages, frameworks, and architectural patterns through targeted modifications to the prompt examples, without requiring changes to the underlying approach.

7.6 Implications for Practitioners

This section provides practical guidance for software engineers, quality assurance professionals, and engineering teams seeking to adopt the RestTSLLM approach in real-world projects. The recommendations below are grounded in the experimental findings of this

[1] https://github.com/Mutcholoko/RESTSLLM-Python-Java

dissertation and the follow-up studies described in Section 7.5.

7.6.1 When to Consider RestTSLLM

The RestTSLLM approach is particularly well-suited for the following scenarios:

- The project exposes a REST API documented by a well-maintained OpenAPI specification that accurately reflects the API's current behavior, including field constraints, required parameters, authentication schemes, and example values.
- The team lacks an existing integration test suite and needs to bootstrap one rapidly—or the existing suite has low coverage and needs to be complemented with additional scenarios.
- The API specification is sufficiently concise (typically fewer than 1,000–1,300 lines of OAS) to fit within the practical processing window of the chosen LLM. For larger specifications, the endpoint-group segmentation strategy described in Chapter 4 is recommended.
- The team has access to some LLM to execute and validate the generated tests.

Conversely, practitioners should be aware of the following situations where the approach may require additional effort: highly complex or poorly documented specifications; APIs with extensive asynchronous or event-driven flows (for which the extension described in Section 7.5.2 is more appropriate); and environments with strict data isolation requirements that prevent running tests against a live API instance.

7.6.2 Selecting an LLM

The choice of LLM has a measurable impact on the quality of the generated tests. Based on the results of this study, the following recommendations apply:

- **For highest quality**: Claude 3.5 Sonnet achieved the best overall results across all metrics—the only model that produced no failed tests—and is the recommended choice when output quality is the primary criterion.
- **For cost-effectiveness**: Deepseek R1, Qwen 2.5 32b, and Sabiá 3 delivered results within 7,5% of the top performer at a fraction of the cost (under $0,09 per project).

These models represent strong choices for budget-constrained settings or large-scale experiments.

- **For multilingual or local contexts**: Sabiá 3 is a Brazilian-developed model optimized for Portuguese and is a viable choice for teams that prefer locally developed, regionally focused models.
- **General expectation**: all eight models evaluated achieved average success rates above 95,5%, indicating that any well-supported general-purpose LLM with strong code generation capabilities is likely to produce usable results.

7.6.3 Adapting the Approach to Other Technologies

One of the key design principles of RestTSLLM is its technology-agnosticism. Practitioners working with languages other than C# can adapt the approach by replacing the code examples in *Prompt 2* (Stage II) with equivalent examples written in the target framework. The remaining stages—OAS interpretation, TSL generation, and the *system prompt* configuration—require no modifications.

As demonstrated by Felix (2025), this adaptation is technically feasible for Java (REST Assured) and Python (PyTest) with straightforward prompt adjustments. Practitioners should ensure that the replacement examples are clear, well-structured, and representative of the coding conventions of the target framework, as the quality of few-shot examples significantly influences output consistency.

7.6.4 Practical Execution Checklist

The following checklist summarizes the recommended steps for applying RestTSLLM to a new project:

1. **Prepare the OAS**: review the specification to ensure it accurately documents all endpoints, parameters, constraints, and authentication requirements. Incomplete or inaccurate specifications are the most common source of test failures (see Subsection 7.2.1).
2. **Segment the specification**: if the OAS contains more than approximately 1,000 lines, organize endpoints into logical groups using OAS tags and process each group independently in a separate execution of *Prompt 3* and *Prompt 4* (Stage III).

3. **Configure the prompts**: use the automation script available in the public repository (BARRADAS, 2025) and configure the target model, technology, and segmentation strategy. If adapting to a language other than C#, replace the code examples in Prompt 2 and Prompt 4 following the guidance in Section 7.6.3.

4. **Run the generation pipeline**: execute the four-prompt pipeline as described in Chapter 4. Store the generated TSL and test code for review.

5. **Review the generated tests**: manually inspect the generated tests against the OAS and known business rules, focusing on: (a) boundary values and property length constraints; (b) authentication flows; (c) required versus optional fields; and (d) JSON deserialization logic. These categories accounted for the majority of failures observed in this study.

6. **Execute and validate**: run the test suite against a live or test-environment instance of the API. Apply reprompting for any tests that fail due to structural or logical errors that can be corrected with targeted follow-up instructions.

7. **Measure and iterate**: collect success rate, coverage, and mutation score to assess the quality of the generated suite. Use areas of low coverage as signals for additional targeted generation cycles.

7.6.5 Cost Expectations

The cost of running the RestTSLLM pipeline is very low. In the experiments conducted for this dissertation, the average cost per project ranged from $0,02 (LLaMA 3.2 90b, open-source, self-hosted) to $0,78 (Deepseek R1), with even the most expensive model (Deepseek R1) remaining well below $1.00 per execution. This makes LLM-based test generation financially accessible even for small teams and frequent re-generation cycles triggered by API updates.

7.6.6 Managing Output Variability

Due to the stochastic nature of LLM generation (see the **Result Randomness** threat in Section 7.4), the same prompt may produce slightly different outputs across runs. Practitioners can reduce this variability by: (a) using a fixed random seed where the chosen model's API supports it; (b) keeping the temperature parameter at or below 1; and (c) validating the generated tests against the OAS immediately after generation, before

incorporating them into the project's test suite. When a generated test is structurally incorrect, targeted reprompting—submitting a follow-up prompt that identifies the specific error and requests a correction—is more effective than discarding and regenerating the full suite.

8 Conclusion

This dissertation proposed RestTSLLM, an approach that combines TSL and LLM to automate the generation of integration tests for REST APIs from OpenAPI Specification. Its effectiveness was assessed through a comparative experimental evaluation, and its acceptance was investigated through a practitioners' survey.

The results directly address the three research questions defined in this study. **Regarding RQ1** (*Are LLMs effective in generating integration tests that reflect the intended business logic and context?*), the comparative experimental evaluation results demonstrated that all eight evaluated LLMs were capable of generating integration tests that reflect the intended business logic of REST APIs, producing compilable, readable, and contextually coherent test cases aligned with the OpenAPI Specification.

Regarding RQ2 (*Which LLM is the most effective in generating integration tests?*), the models Claude 3.5 Sonnet, Deepseek R1, Qwen 2.5 32b, and Sabiá 3 achieved the best overall performance, with Claude 3.5 Sonnet standing out as the top-performing model across all metrics, being the only model that produced no faulty tests.

Regarding RQ3 (*What is the level of acceptance of the RestTSLLM approach among practitioners involved in the REST API development lifecycle?*), the acceptance survey with 96 practitioners revealed a high level of acceptance: an overall agreement of 4.28 on a 1–5 scale, a strong intention to adopt the approach, and particular appreciation for its perceived usefulness and the intermediate TSL representation. The main reservation was trust in the generated tests, which practitioners conditioned on retaining a human-in-the-loop review — a nuance consistently observed across the quantitative and qualitative evidence.

These results reinforce both the feasibility of using LLMs to automate integration testing for REST APIs and its acceptance among practitioners, highlighting the potential of prompt-based strategies combined with intermediate TSL generation to guide LLMs more effectively.

In summary, this dissertation contributes by proposing a reusable approach – RestTSLLM – for LLM-based test generation supported by prompt engineering techniques. The comparison of multiple LLMs provided valuable insights into their relative effectiveness, while the development of an automated multi-model execution script lays the groundwork for future, large-scale experimentation across diverse models and configurations. Additionally, the acceptance survey provides empirical evidence regarding RQ3, assessing the perceived utility and adoption intention of the approach among professionals in the REST API development lifecycle. The relevance of these contributions is further supported by their peer-reviewed publication: the core of this work was published as a full paper at the 39th Brazilian Symposium on Software Engineering (SBES 2025) (BARRADAS; PAES; NEVES, 2025).

Future work includes addressing the limitations discussed throughout this study, exploring ways to strengthen and broaden the applicability of the proposed approach. As described in Section 7.5, initial steps in this direction have already been taken: Felix (2025) extended RestTSLLM to Java and Python, confirming the language independence of the TSL stage, and, in an ongoing undergraduate thesis, Calvet (2026) is investigating the extension of the pipeline to asynchronous and event-driven REST APIs. These works establish a foundation from which several research directions can be pursued.

Broader empirical evaluation. We plan to conduct broader experiments to further validate the generalizability of the method. This includes testing with other emerging LLM models and re-evaluating the approach with updated versions of those already assessed – as model capabilities evolve rapidly and the versions used here reflect availability during the first half of 2025 (see Section 7.4) –, expanding evaluations to more complex and large-scale REST APIs with pre-existing test suites to enable quantitative comparisons, and investigating the use of additional languages in prompts and examples — beyond Portuguese and English — to evaluate multilingual performance. We also plan to compare the time and effort of manual test creation versus LLM-assisted generation, and to broaden the evaluation beyond success rate, coverage, and mutation score to capture complementary quality aspects of the generated tests, such as readability, maintainability, runtime performance, and execution cost.

Ablation study. The current evaluation assesses the RestTSLLM pipeline as a whole; the individual contribution of each design decision remains to be quantified. A systematic ablation would compare: (1) the full two-step pipeline (OAS → TSL → tests) against a direct one-step baseline (OAS → tests, bypassing the TSL intermediate representation);

(2) few-shot prompting against zero-shot prompting; and (3) the decomposed multi-prompt strategy against a single monolithic prompt. Such an experiment would provide causal evidence for the role of the TSL layer and the few-shot strategy in the observed results, strengthening the methodological justification for the approach's design.

Extending the approach. We intend to broaden the range of systems and inputs the approach can handle. This includes consolidating the extensions to Java, Python, and event-driven architectures into a unified, fully automated pipeline; improving support for outdated or incomplete API specifications; mitigating LLM token limitations in scenarios involving long prompts; and incorporating alternative input formats beyond the OpenAPI Specification (OAS) — in particular, using the system's source code as a complementary input to enrich test generation and to overcome the inherent limitations of a purely black-box, specification-driven perspective (Sections 7.4 and 7.5.1). We also plan to provide first-class support for test-data provisioning, such as synthetic data generation, fakers, and database seeding, to enable the approach in realistic environments.

Tooling and workflow integration. We also intend to package the approach for everyday use and reduce the residual manual effort. This includes automatic handling and correction of errors through reprompting techniques, as well as self-healing or agent-based validation of the generated tests; combining multiple generated tests to optimize coverage and fault detection; introducing an intermediate review checkpoint over the generated TSL before the executable code is produced; implementing IDE plugins and CI/CD integration with coverage reporting to better fit the approach into developers' workflows; providing 'skills' and agents to facilitate the use of prompt engineering in modern LLMs that support these extensions; and supporting test maintenance and evolution throughout the software lifecycle, enabling incremental updates. Notably, several of these directions — in particular the intermediate TSL review checkpoint, CI/CD integration with coverage reporting, self-healing or agent-based validation, and first-class test-data provisioning — were independently requested by the surveyed practitioners (Section 7.3.8), which reinforces their practical relevance.

Evolving the acceptance study. Finally, regarding the acceptance study, future work includes expanding the survey to a larger and more geographically diverse sample, conducting longitudinal follow-up to assess actual adoption after tool availability, and correlating survey responses with objective usage metrics in real development environments. A larger sample would also enable formal hypothesis testing to assess whether the observed differences across professional profiles (role, seniority, and technology stack) reach statistical

significance, complementing the exploratory subgroup analysis reported here.

AI Usage Disclaimer

ChatGPT-5.5 was used to assist with the organization and revision of the manuscript and speed up the writing of LaTeX code. We carefully examined and often corrected AI suggestions, whereby we took full responsibility for the form and content of the dissertation.

REFERENCES

ABONIZIO, Hugo et al. **Sabiá-3 Technical Report**. [S. l.: s. n.], 2024. Available from: <https://arxiv.org/abs/2410.12049>.

ACM SIGSOFT EMPIRICAL STANDARDS. **Questionnaire Surveys — Empirical Standards for Software Engineering Research**. [S. l.: s. n.], 2020. https://www2.sigsoft.org/EmpiricalStandards/docs/. Accessed on: May 24, 2026.

ALMEIDA, Thales Sales et al. **Sabiá-2: A New Generation of Portuguese Large Language Models**. [S. l.: s. n.], 2024. Available from: <https://arxiv.org/abs/2403.09887>.

ALSHAHWAN, Nadia; HARMAN, Mark; MARGINEAN, Alexandru. Software Testing Research Challenges: An Industrial Perspective. In: 2023 IEEE Conference on Software Testing, Verification and Validation. [S. l.: s. n.], 2023. P. 1–10. Available from: <https://doi.org/10.1109/ICST57152.2023.00008>.

ANAND, Abhineet; UDDIN, Azeem. Importance of software testing in the process of software development. **International Journal for Scientfic Research and Development (IJSRD)**, 2019. Available from: <https://www.researchgate.net/publication/331223692>.

ARCURI, Andrea. RESTful API automated test case generation with EvoMaster. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 28, n. 1, p. 1–37, 2019. DOI: 10.1145/3293455.

AYDIN, Omer et al. **Generative AI in Academic Writing: A Comparison of DeepSeek, Qwen, ChatGPT, Gemini, Llama, Mistral, and Gemma**. [S. l.: s. n.], 2025. Available from: <https://arxiv.org/abs/2503.04765>.

BARRADAS, Thiago. **Integration Test Generation - LLM Efficiency - Repository**. [S. l.: s. n.], 2025. Accessed on: June 26, 2025. Available from: <https://github.com/uffsoftwaretesting/RestTSLLM>.

BARRADAS, Thiago; PAES, Aline; NEVES, Vania. Combining TSL and LLM to Automate REST API Testing: A Comparative Study. In: PROCEEDINGS of the 39th Brazilian Symposium on Software Engineering (SBES). Recife/PE: SBC, 2025. P. 71–81. DOI: 10.5753/sbes.2025.9670. Available from: <https://sol.sbc.org.br/index.php/sbes/article/view/36987>.

BASILI, Victor R.; CALDIERA, Gianluigi; ROMBACH, Hans Dieter. The Goal Question Metric Approach. In: ENCYCLOPEDIA of Software Engineering. [S. l.]: sn, 1994. P. 528–532. Available from: <https://api.semanticscholar.org/CorpusID:13884048>.

BELZNER, Lenz; GABOR, Thomas; WIRSING, Martin. Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study. In: AISOLA. [S. l.: s. n.], 2023. Available from: <https://doi.org/10.1007/978-3-031-46002-9_23>.

BOTPRESS. **Discover models by Popularity**. [S. l.: s. n.], 2025. Accessed on: January 25, 2025. Available from: <https://botpress.com/llm-ranking>.

BOUKHLIF, Mohamed; KHARMOUM, Nassim; HANINE, Mohamed. LLMS for intelligent software testing: a comparative study. In: PROCEEDINGS of the 7th International Conference on Networking, Intelligent Systems and Security. [S. l.: s. n.], 2024. Available from: <https://dl.acm.org/doi/10.1145/3659677.3659749>.

CALVET, Leonardo. **Extending RestTSLLM to Asynchronous and Event-Driven REST APIs**. Niterói, RJ, Brazil: [s. n.], 2026. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação), Universidade Federal Fluminense, Instituto de Computação. In preparation. Advisor: Prof. Vânia de Oliveira Neves.

CAO, Clinton; PANICHELLA, Annibale; VERWER, Sicco. Automated Test-Case Generation for REST APIs Using Model Inference Search Heuristic. In: 2025 IEEE/ACM International Conference on Automation of Software Test (AST). [S. l.: s. n.], 2025. P. 29–40. DOI: 10.1109/AST66626.2025.00010.

CHANG, Hung-Fu; SHIRAZI, Mohammad Shokrolah. A Systematic Approach for Assessing Large Language Models' Test Case Generation Capability. **Software**, MDPI, v. 4, n. 5, 2025. DOI: 10.3390/software4010005. Available from: <https://www.mdpi.com/2674-113X/4/1/5>.

CORRADINI, Davide et al. **Empirical Comparison of Black-box Test Case Generation Tools for RESTful APIs**. [S. l.: s. n.], 2021. Available from: <https://arxiv.org/abs/2108.08196>.

CRUZES, Daniela S.; DYBÅ, Tore. Recommended Steps for Thematic Synthesis in Software Engineering. In: PROCEEDINGS of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM). [S. l.]: IEEE, 2011. P. 275–284. DOI: 10.1109/ESEM.2011.36.

DAVIS, Fred D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. **MIS quarterly**, Management Information Systems Research Center, University of Minnesota, v. 13, n. 3, p. 319–340, 1989. Available from: <https://dl.acm.org/doi/abs/10.2307/249008>.

DEFINITION, This is. **What's the most popular LLM?** [S. l.: s. n.], 2025. Accessed on: January 20, 2025. Available from: <https://www.thisisdefinition.com/insights/most-popular-llm>.

DELAMARO, Marcio; JINO, Mario; MALDONADO, Jose. **Introdução ao teste de software**. [S. l.]: Elsevier Brasil, 2013.

ERDAL, Hasan. **Shortener API - Repository**. [S. l.: s. n.], 2025. Accessed on: March 24, 2025. Available from: <https://github.com/Filiphasan/dotnet-minify-url>.

FAN, Angela et al. Large Language Models for Software Engineering: Survey and Open Problems. In: 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). [S. l.: s. n.], 2023. Available from: <https://arxiv.org/abs/2310.03533>.

FELIX, Gabriel Gavazzi. **De Especificação OpenAPI para testes Java e Python: Utilizando LLMs para Geração de Testes de APIs REST**. Niterói, RJ, Brazil: [s. n.], 2025. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação), Universidade Federal Fluminense, Instituto de Computação. Orientadora: Profa. Vânia de Oliveira Neves. Available from: <https://app.uff.br/riuff/handle/1/43014>.

FIELDING, Roy Thomas. **Architectural styles and the design of network-based software architectures**. [S. l.]: University of California, Irvine, 2000.

FORTUNE BUSINESS INSIGHTS. **Dot Net Development Service Market Size, Share, Growth**. Accessed on: January 12, 2025. 2023. Available from: <https://www.fortunebusinessinsights.com/dot-net-development-service-market-111361>.

FOWLER, David. **Todo API - Repository**. [S. l.: s. n.], 2025. Accessed on: March 24, 2025. Available from: <https://github.com/davidfowl/TodoApp>.

FREITAG, Raquel M. Ko; GOIS, Túlio Sousa de. Performance in a dialectal profiling task of LLMs for varieties of Brazilian Portuguese. In: ANAIS do XV Simpósio Brasileiro de Tecnologia da Informação e da Linguagem Humana (STIL 2024). [S. l.]: Sociedade Brasileira de Computação, 2024. (STIL 2024), p. 317–326. Available from: <http://dx.doi.org/10.5753/stil.2024.241891>.

GLASER, Barney G.; STRAUSS, Anselm L. **The Discovery of Grounded Theory: Strategies for Qualitative Research**. Chicago, IL, USA: Aldine, 1967.

GOLMOHAMMADI, Amid; ZHANG, Man; ARCURI, Andrea. Testing RESTful APIs: A Survey. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, USA, v. 33, n. 1, p. 1–41, 2023. Available from: <https://doi.org/10.1145/3617175>.

GOMES, Evandro. **Supermarket API - Repository**. [S. l.: s. n.], 2025. Accessed on: March 24, 2025. Available from: <https://github.com/evgomes/supermarket-api>.

GU, Sijia; NASHID, Noor; MESBAH, Ali. **LLM Test Generation via Iterative Hybrid Program Analysis**. [S. l.: s. n.], 2025. arXiv: 2503.13580 [cs.SE].

HAGOS, Desta Haileselassie; BATTLE, Rick; RAWAT, Danda B. **Recent Advances in Generative AI and Large Language Models: Current Status, Challenges, and Perspectives**. [S. l.: s. n.], 2024. Available from: <https://arxiv.org/abs/2407.14962>.

HOU, Xinyi et al. Large language models for software engineering: A systematic literature review. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, v. 33, n. 8, p. 1–79, 2024. Available from: <https://dl.acm.org/doi/full/10.1145/3695988>.

JAIN, Kush et al. **Mind the Gap: The Difference Between Coverage and Mutation Score Can Guide Testing Efforts**. [S. l.: s. n.], 2023. Available from: <https://arxiv.org/abs/2309.02395>.

JINDAL, Tanu. Importance of Testing in SDLC. **International Journal of Engineering and Applied Computer Science (IJEACS)**, 2016. Available from: <https://www.researchgate.net/publication/312041152>.

JOSHI, Ankur et al. Likert Scale: Explored and Explained. **British Journal of Applied Science & Technology**, v. 7, p. 396–403, Jan. 2015. DOI: 10.9734/BJAST/2015/14975.

KHOT, Tushar et al. Decomposed Prompting: A Modular Approach for Solving Complex Tasks. In: THE Eleventh International Conference on Learning Representations. [S. l.: s. n.], 2023. Available from: <https://arxiv.org/abs/2210.02406>.

KIM, Myeongsoo; SINHA, Saurabh; ORSO, Alessandro. Llamaresttest: Effective rest api testing with small language models. **Proceedings of the ACM on Software Engineering**, ACM New York, NY, USA, v. 2, FSE, p. 465–488, 2025. Available from: <https://doi.org/10.1145/3715737>.

KIM, Myeongsoo; STENNETT, Tyler; SHAH, Dhruv, et al. Leveraging Large Language Models to Improve REST API Testing. In: 2024 IEEE/ACM 46th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). [S. l.]: ACM, 2024. P. 37–41. Available from: <https://doi.org/10.1145/3639476.3639769>.

KIM, Myeongsoo; STENNETT, Tyler; SINHA, Saurabh, et al. **A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs**. [S. l.: s. n.], 2025. arXiv: 2411.07098 [cs.SE]. Available from: <https://arxiv.org/abs/2411.07098>.

KITCHENHAM, Barbara A; PFLEEGER, Shari L. Personal opinion surveys. In: GUIDE to advanced empirical software engineering. London: Springer, 2008. P. 63–92. DOI: 10.1007/978-1-84800-044-5_3.

KOGLER, Leon; EHRHART, Maximilian, et al. **RESTifAI: LLM-Based Workflow for Reusable REST API Testing**. [S. l.: s. n.], 2025. ICSE 2026 Demo Track. arXiv: 2512.08706 [cs.SE]. Available from: <https://arxiv.org/abs/2512.08706>.

KOGLER, Leon; HANGLER, Stefan, et al. **RESTestBench: A Benchmark for Evaluating the Effectiveness of LLM-Generated REST API Test Cases from NL Requirements**. [S. l.: s. n.], 2026. arXiv: 2604.25862 [cs.SE]. Available from: <https://arxiv.org/abs/2604.25862>.

KOZERA, Jakub. **Restaurants API - Repository**. [S. l.: s. n.], 2025. Accessed on: March 24, 2025. Available from: <https://github.com/jakubkozera/Restaurants>.

LE, Tri et al. KAT: Dependency-Aware Automated API Testing with Large Language Models. In: 2024 IEEE Conference on Software Testing, Verification and Validation (ICST). [S. l.]: IEEE, May 2024. P. 82–92. DOI: 10.1109/icst60714.2024.00017. Available from: <http://dx.doi.org/10.1109/ICST60714.2024.00017>.

LERCHER, Alexander. Managing API Evolution in Microservice Architecture. In: PROCEEDINGS of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings. [S. l.]: ACM, 2024. P. 195–197. Available from: <https://doi.org/10.1145/3639478.3639800>.

LIU, Jing et al. **Can LLM Generate Regression Tests for Software Commits?** [S. l.: s. n.], 2025. arXiv: 2501.11086 [cs.SE]. Available from: <https://arxiv.org/abs/2501.11086>.

LIVEBENCH. **LiveBench - Leaderboard**. [S. l.: s. n.], 2024. Accessed on: January 25, 2025. Available from: <https://livebench.ai/#/>.

MENDOZA, Isela et al. Comparative Analysis of Large Language Model Tools for Automated Test Data Generation from BDD. In: ANAIS do XXXVIII Simpósio Brasileiro de Engenharia de Software. Curitiba/PR: SBC, 2024. Available from: <https://sol.sbc.org.br/index.php/sbes/article/view/30369>.

MICROSOFT. **Integration tests in ASP.NET Core**. Accessed on: April 06, 2025. 2025. Available from: <https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests>.

MU, Norman et al. **A Closer Look at System Prompt Robustness**. [S. l.: s. n.], 2025. Available from: <https://arxiv.org/abs/2502.12197>.

NEBULY. **LLM System Prompt vs. User Prompt**. Accessed on: April 07, 2025. 2024. Available from: <https://www.nebuly.com/blog/llm-system-prompt-vs-user-prompt>.

NEWMAN, Sam. **Building Microservices: Designing Fine-Grained Systems**. [S. l.]: O'Reilly Media, 2015. ISBN 9781491950357.

OLSTHOORN, Mitchell. More Effective Test Case Generation with Multiple Tribes of AI. In: PROCEEDINGS of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. [S. l.]: ACM, 2022. P. 286–290. Available from: <https://doi.org/10.1145/3510454.3517066>.

OPENROUTER. **LLM Rankings - Programming**. [S. l.: s. n.], 2025. Accessed on: January 25, 2025. Available from: <https://openrouter.ai/rankings/programming?view=month>.

ORSO, Alessandro; ROTHERMEL, Gregg. Software testing: a research travelogue (2000-2014). In: FUTURE of Software Engineering Proceedings. [S. l.]: Georgia Institute of Technology, 2014. Available from: <https://dl.acm.org/doi/10.1145/2593882.2593885>.

OSTRAND, Thomas J.; BALCER, Marc J. The category-partition method for specifying and generating functional tests. **Communications of the ACM**, ACM New York, NY, USA, v. 31, n. 6, p. 676–686, 1988. Available from: <https://dl.acm.org/doi/10.1145/62959.62964>.

OUÉDRAOGO, Wendkûuni C. et al. **Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation**. [S. l.: s. n.], 2024. Available from: <https://arxiv.org/abs/2407.00225>.

PEREIRA, André; LIMA, Bruno; FARIA, João Pascoal. APITestGenie: Generating Web API Tests from Requirements and API Specifications with LLMs. **arXiv preprint arXiv:2604.02039**, 2026. Available from: <https://arxiv.org/pdf/2604.02039>.

PEZZÈ, Mauro et al. The Trailer of the ACM 2030 Roadmap for Software Engineering. **ACM SIGSOFT Software Engineering Notes**, ACM New York, NY, USA, v. 49, n. 4, p. 31–40, 2024. Available from: <https://doi.org/10.1145/3696117.3696126>.

PIRES, Ramon et al. Sabiá: Portuguese Large Language Models. In: INTELLIGENT Systems. [S. l.]: Springer Nature Switzerland, 2023. P. 226–240. Available from: <http://dx.doi.org/10.1007/978-3-031-45392-2_15>.

QIU, Ruizhong et al. **How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark**. [S. l.: s. n.], 2024. Available from: <https://arxiv.org/abs/2406.06647>.

RALPH, Paul et al. **Empirical Standards for Software Engineering Research**. [S. l.: s. n.], 2020. arXiv: 2010.03525 [cs.SE]. Available from: <https://arxiv.org/abs/2010.03525>.

REILE, Christoph et al. Bunk8s: Enabling Easy Integration Testing of Microservices in Kubernetes. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). [S. l.: s. n.], 2022. P. 459–463. DOI: 10.1109/SANER53432.2022.00062.

SADOWSKI, Albert; CHUDZIAK, Jarosław A. Structured Decomposition for LLM Reasoning: Cross-Domain Validation and Semantic Web Integration. **arXiv e-prints**, arXiv:2601.01609, arxiv:2601.01609, Jan. 2026. DOI: 10.48550/arXiv.2601.01609. arXiv: 2601.01609 [cs.AI].

SALDAÑA, Johnny. **The Coding Manual for Qualitative Researchers**. 2nd. Thousand Oaks, CA, USA: SAGE Publications, 2013.

SHAKUDO. **Top 9 Large Language Models as of January 2025**. [S. l.: s. n.], 2025. Accessed on: January 26, 2025. Available from: <https://www.shakudo.io/blog/top-9-large-language-models>.

SIDDIQ, Mohammed Latif et al. **Large Language Models for Software Engineering: A Reproducibility Crisis**. [S. l.: s. n.], 2025. arXiv: 2512.00651 [cs.SE]. Available from: <https://arxiv.org/abs/2512.00651>.

SOMMERVILLE, Ian. **Software Engineering**. 10. ed. [S. l.]: Pearson, 2016. ISBN 9781292096131.

SOTIRIADIS, Stelios et al. Unit and Integration Testing of Modular Cloud Services. In: 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA). [S. l.: s. n.], 2017. Available from: <https://doi.org/10.1109/AINA.2017.57>.

SOYSA, Poorna. **Books API - Repository**. [S. l.: s. n.], 2025. Accessed on: March 24, 2025. Available from: <https://github.com/poorna-soysa/books-api-docker-compose-postgresql-redis>.

STRYKER. **Stryker .NET - Configuration**. Accessed on: April 09, 2025. 2025. Available from: <https://stryker-mutator.io/docs/stryker-net/configuration/>.

TRINKENREICH, Bianca et al. An Empirical Investigation on the Challenges Faced by Women in the Software Industry: A Case Study. In: PROCEEDINGS of the 44th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS). [S. l.]: ACM, 2022. P. 24–35. DOI: 10.1145/3510458.3513018.

TU, Haoxin et al. **Cottontail: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation**. [S. l.: s. n.], 2025. arXiv: 2504.17542 [cs.SE]. Available from: <https://arxiv.org/abs/2504.17542>.

TUTEJA, Maneela; DUBEY, Gaurav, et al. A research study on importance of testing and quality assurance in software development life cycle (SDLC) models. **International Journal of Soft Computing and Engineering (IJSCE)**, IJSCE, v. 2, n. 3, p. 251–257, 2012. Available from: <https://www.ijsce.org/portfolio-item/c0761062312/>.

UNITE.AI. **Best Large Language Models (LLMs) in 2025**. [S. l.: s. n.], 2025. Accessed on: January 20, 2025. Available from: <https://www.unite.ai/best-large-language-models-llms/>.

VELLUM. **LLM Leaderboard - Model Comparison**. [S. l.: s. n.], 2025. Accessed on: January 26, 2025. Available from: <https://www.vellum.ai/llm-leaderboard>.

VIGLIANISI, Emanuele; DALLAGO, Michael; CECCATO, Mariano. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In: IEEE. 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). [S. l.: s. n.], 2020. Available from: <https://doi.org/10.1109/ICST46399.2020.00024>.

WANG, Junjie et al. **Software testing with large language models: Survey, landscape, and vision**. v. 50. [S. l.]: IEEE, 2024. P. 911–936. DOI: 10.1109/TSE.2024.3368208. Available from: <https://doi.org/10.1109/TSE.2024.3368208>.

WEI, Chenhao et al. How do developers structure unit test cases? an empirical analysis of the aaa pattern in open source projects. **IEEE Transactions on Software Engineering**, IEEE, v. 51, n. 4, p. 1007–1038, 2025.

WEI, Jason et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In_____. **Advances in Neural Information Processing Systems**. [S. l.]: Curran Associates, Inc., 2022. v. 35, p. 24824–24837. Available from: <https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf>.

WILLIAMS, Trevoir. **Hotels API - Repository**. [S. l.: s. n.], 2025. Accessed on: March 24, 2025. Available from: <https://github.com/trevoirwilliams/HotelListing.API.NET>.

WOHLIN, Claes et al. **Experimentation in software engineering**. [S. l.]: Springer, 2012. v. 236. DOI: 10.1007/978-3-642-29044-2.

XIA, Chunqiu Steven; DENG, Yinlin; ZHANG, Lingming. **Top Leaderboard Ranking = Top Coding Proficiency, Always? EvoEval: Evolving Coding Benchmarks via LLM**. [S. l.: s. n.], 2024. Available from: <https://arxiv.org/abs/2403.19114>.

YAO, Shunyu et al. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In_____. **Advances in Neural Information Processing Systems**. [S. l.]: Curran Associates, Inc., 2023. v. 36, p. 11809–11822. Available from: <https://proceedings.neurips.cc/paper_files/paper/2023/file/271db9922b8d1f4dd7aaef84ed5ac703-Paper-Conference.pdf>.

YUAN, Zhiqiang et al. Evaluating and Improving ChatGPT for Unit Test Generation. **Proceedings of the ACM on Software Engineering**, ACM, New York, NY, USA, v. 1, FSE, article 76, 2024. DOI: 10.1145/3660783. Available from: <https://doi.org/10.1145/3660783>.

ZAPIER. **The best large language models (LLMs) in 2025**. [S. l.: s. n.], 2025. Accessed on: January 20, 2025. Available from: <https://zapier.com/blog/best-llm/>.

ZHANG, Lechen et al. **SPRIG: Improving Large Language Model Performance by System Prompt Optimization**. [S. l.: s. n.], 2024. Available from: <https://arxiv.org/abs/2410.14826>.

ZHANG, Peng et al. **Test suite effectiveness metric evaluation: what do we know and what should we do?** [S. l.: s. n.], 2022. Available from: <https://arxiv.org/abs/2204.09165>.

ZHANG, Quanjun et al. A survey on large language models for software engineering. **Science China Information Sciences**, Springer, v. 69, n. 4, p. 141102, 2026. DOI: 10.1007/s11432-025-4670-0. Available from: <https://link.springer.com/article/10.1007/s11432-025-4670-0>.

ZHENG, Zibin et al. Towards an understanding of large language models in software engineering tasks. **Empirical Software Engineering**, Springer, v. 30, n. 50, 2025. DOI: 10.1007/s10664-024-10602-0. Available from: <https://link.springer.com/article/10.1007/s10664-024-10602-0>.

Glossary

.NET Cross-platform framework and runtime by Microsoft for building applications; used here for REST APIs and testing with xUnit

agent An LLM-based autonomous system that perceives an environment, plans sequences of actions, and executes them—typically through tool use (e.g., web search, code execution, API calls)—to accomplish a goal with minimal human intervention. Agents extend single-turn prompt engineering into multi-step, goal-directed workflows, and are increasingly used in software engineering automation tasks such as test generation and debugging

APITestGenie Tool that generates executable REST API integration tests from business requirements in natural language and OpenAPI Specifications, using LLMs and Retrieval-Augmented Generation (RAG)

ARTE Automated REST API testing tool that generates and executes test cases from API specifications (e.g., OpenAPI), aiming to improve coverage and fault detection in black-box testing

AutoRestTest Multi-agent REST API testing approach that combines LLMs, a Semantic Property Dependency Graph, and Multi-Agent Reinforcement Learning to optimize runtime exploration and fault detection

bBOXRT Black-box REST API testing tool that generates and executes requests from API specifications to explore behaviors across endpoints and detect failures

black-box testing Testing technique that evaluates system functionality based on inputs and outputs without knowledge of internal implementation

Category–Partition method Test design technique that derives test cases by partitioning input domains into categories and choices, and combining them systematically

chain-of-thoughts Technique that elicits step-by-step intermediate reasoning in natural language before producing the final answer, improving correctness on multi-step problems

coverage In testing, the proportion of elements exercised by a test suite; depending on the criterion, may refer to code coverage (statements, branches, paths) or requirement/scenario coverage (requirements, behaviors, endpoints)

decomposed prompting A prompt engineering strategy that breaks a complex task into a sequence of subtasks or intermediate steps (e.g., extract requirements → specify test cases → generate test code), reducing cognitive load and improving quality and reproducibility

endpoint An individual operation exposed by a REST API, identified by an HTTP method and a URL path (e.g., `POST /users`), through which clients interact with a specific resource or action

EvoMaster Search-based automated testing tool for REST API that uses evolutionary algorithms to generate and optimize test inputs and operation sequences; supports both black-box mode (from OAS only) and white-box mode (with runtime source-code instrumentation), targeting fault detection and code coverage

few-shot A prompt engineering technique in which the model is given a small number of labeled examples ("shots") in the prompt to induce the desired pattern without additional training

fine-tuned Model adapted from a pretrained base by additional training on task or domain-specific data (e.g., full fine-tuning or parameter-efficient methods) to improve performance

generative AI A class of artificial intelligence models capable of producing new content—such as text, code, images, or audio—by learning statistical patterns from large training corpora; LLMs are the most prominent example in the domain of text and code generation

integration test Test that verifies the interactions between components, services, or systems (e.g., via REST APIs), focusing on information exchange and cross-boundary behavior; contrasts with unit tests that isolate components

KAT Dependency-aware automated REST API testing approach that constructs an Operation Dependency Graph from the OpenAPI Specification and uses GPT to generate operation sequences, test scripts, and test data

Likert scale A psychometric rating scale in which respondents indicate their level of agreement with a statement, typically on a five-point ordinal range (e.g., 1 = Strongly Disagree to 5 = Strongly Agree); widely used in survey research to measure attitudes, perceptions, and behavioral intentions

LlamaREST-EX Specialized model within LlamaRestTest for generating valid inputs that satisfy detected constraints and dependencies

LlamaREST-IPD Specialized model within LlamaRestTest for identifying input-parameter dependencies in REST APIs

LlamaRestTest Black-box REST API testing approach based on fine-tuned and quantized LLaMA 3 models that adapts test inputs from server feedback to detect parameter dependencies and generate valid requests

MIT License The MIT License is a permissive open-source software license that allows users to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software with very few restrictions

mutation score Proportion of artificial faults (mutants) killed by the test suite, used as a proxy for test effectiveness

NLP2REST Automated REST API test-generation tool that leverages natural-language processing over API documentation and specifications (e.g., OpenAPI) to derive inputs, constraints, and requests for exercising endpoints

OpenAPI Specification for describing REST APIs in a standardized, and machine-readable format

prompt Textual instruction provided to a LLM that specifies the task, context, constraints, and/or examples to elicit the desired output

prompt engineering A set of techniques for designing instructions, examples, and constraints to steer LLMs toward correct, useful, and safe outputs

prompting The practice of eliciting model behavior by composing and issuing prompts (instructions, context, and examples) to achieve a target output; encompasses approaches such as zero-shot, few-shot, chain-of-thoughts, tree-of-thoughts, and decomposed prompting

quantized Model with weights/activations stored in lower numeric precision (e.g., 8-bit, 4-bit) to reduce memory and latency, typically with a small accuracy trade-off

reprompting Iteratively refining a model's output by issuing follow-up prompts that add constraints, corrections, or feedback to fix errors, expand truncated outputs, or obtain missing parts

RESTest Automated testing framework for REST APIs that uses API specifications to generate functional test cases and evaluate service responses against expected behaviors

RESTestBench Benchmark for evaluating LLM-generated REST API test cases, distinguishing non-refinement from refinement-based approaches and introducing requirements-oriented mutation testing as an evaluation strategy

RESTGPT Approach that uses a language model to enrich OpenAPI specifications by extracting constraints, rules, and examples to produce a more detailed, machine-readable specification; the enhanced spec can then be consumed by automated REST API test-generation tools. RESTGPT focuses on specification enrichment rather than directly generating tests

RESTifAI LLM-based workflow for generating reusable and CI/CD-ready REST API tests from OpenAPI Specifications, constructing happy-path scenarios and deriving negative cases via server feedback

RESTTestGen Automated REST API testing tool that derives black-box test cases from OpenAPI Specification to exercise endpoints, parameters, and workflows, aiming to improve coverage and fault detection

RestTSLLM Approach proposed in this dissertation that combines TSL and LLMs to automate REST API integration test generation via a two-step pipeline (TSL $\rightarrow$ executable tests)

schema-based fuzzing Fuzzing guided by a formal schema (e.g., OpenAPI), which constrains and structures generated inputs to improve validity and fault discovery in REST API testing

skill An extension mechanism supported by modern LLMs platforms that allows a model to be configured with a specific behavioral profile, tool set, or knowledge base for a particular task. In the context of prompt engineering, a skill encapsulates reusable instructions and examples that can be selectively activated to guide model behavior without modifying the underlying model weights

Stryker.NET Stryker.NET is an automated mutation testing tool for .NET projects that assesses and improves the quality of your test suites by introducing small "mutations" or bugs into your source code and checking if your tests correctly identify them.

success rate The proportion of tests that successfully pass all assertions when run against the target REST APIs; computed as $SR = \frac{\#\,\text{passing tests}}{\#\,\text{executed tests}}$

system prompt Control instruction that configures the model's behavior and constraints for subsequent interactions

test oracle Mechanism or specification used to determine whether the observed outcomes of a test are correct

tree-of-thoughts Reasoning framework that explores multiple reasoning branches as a search tree, scoring and selecting promising trajectories to reach better solutions

unit test Test that evaluates an individual function, method, or class in isolation from the rest of the system, typically replacing external dependencies with test doubles (mocks, stubs, or fakes) to ensure failures are caused solely by the unit under examination; contrasts with integration test, which exercises real interactions between components

user prompt Task instruction authored by the user, containing the query, context, or data to be processed by the model

Visual Studio Visual Studio is a highly popular and comprehensive Integrated Development Environment (IDE) developed by Microsoft, and it is indeed considered the official IDE for developing applications within the .NET ecosystem.

white-box testing Testing technique that evaluates system behavior with full knowledge of the internal implementation, using source-code structure (e.g., control flow, branch conditions) to guide test generation and measure structural coverage

xUnit Unit testing framework for .NET within the xUnit family

zero-shot Technique in which the model receives only the task instruction (no labeled examples) and solves it based on prior knowledge

APPENDIX A – LLMs Mentions By Source

This appendix lists the sources in which each model is mentioned and reports a simple popularity score as the sum of mentions across sources. The selection of sources mixes academic and practitioner outlets to reflect both research and industry visibility.

Table 11 lists the included LLMs (GPT, Llama, Claude, Gemini, DeepSeek, Mixtral, and Qwen). Table 12 lists the excluded LLMs with at least two citations (Grok, Command, StarCoder, and PaLM). LLMs with exactly one citation in any source are omitted (Phi, Nova, InCoder, Copilot, and others).

Table 11: Mentions by source - Included LLMs

Source	GPT	Llama	Claude	Gemini	DeepSeek	Mixtral	Qwen
Hagos, Battle, and Rawat (2024)	✓	✓	✗	✓	✗	✗	✗
Qiu et al. (2024)	✓	✓	✓	✗	✗	✓	✗
Xia, Deng, and Zhang (2024)	✓	✓	✓	✓	✓	✓	✓
Aydin et al. (2025)	✓	✓	✗	✓	✓	✓	✓
Shakudo (2025)	✓	✓	✓	✓	✓	✓	✓
Definition (2025)	✓	✓	✓	✓	✗	✓	✗
Unite.AI (2025)	✓	✗	✓	✓	✓	✗	✗
Zapier (2025)	✓	✓	✓	✓	✓	✓	✓
OpenRouter (2025)	✓	✓	✓	✓	✓	✓	✓
Vellum (2025)	✓	✓	✓	✗	✓	✗	✓
BotPress (2025)	✓	✓	✓	✓	✓	✓	✓
LiveBench (2024)	✓	✓	✓	✓	✓	✓	✓
Total (popularity score)	**12**	**11**	**10**	**10**	**9**	**9**	**8**

Table 12: Mentions by source - Excluded LLMs

Source	Grok	Command	StarCoder	Palm
Hagos, Battle, and Rawat (2024)	✗	✗	✗	✓
Qiu et al. (2024)	✗	✗	✓	✗
Xia, Deng, and Zhang (2024)	✗	✗	✓	✓
Aydin et al. (2025)	✗	✗	✗	✗
Shakudo (2025)	✓	✓	✗	✗
Definition (2025)	✗	✗	✗	✗
Unite.AI (2025)	✓	✗	✗	✗
Zapier (2025)	✓	✓	✗	✗
OpenRouter (2025)	✗	✗	✗	✗
Vellum (2025)	✗	✗	✗	✗
BotPress (2025)	✗	✗	✗	✗
LiveBench (2024)	✗	✗	✗	✗
Total (popularity score)	**3**	**2**	**2**	**2**

APPENDIX B – All Collected Metrics From Prompt Executions

Metrics collected from the execution of all prompts for all projects and all LLMs. Each table represents a project. Table 13 for *todo-api*, Table 14 for *restaurants-api*, Table 15 for *shortener-api*, Table 16 for *books-api*, Table 17 for *hotels-api*, and Table 18 for *supermarket-api*. For simplification purposes, an acronym was used to represent each metric:

- $TC \rightarrow$ Total Cost (USD)
- $T \rightarrow$ Number of Tests
- $FT \rightarrow$ Failed Tests
- $PT \rightarrow$ Passed Tests
- $SR \rightarrow$ Success Rate
- $C \rightarrow$ Coverage
- $M \rightarrow$ Mutation Score
- $S \rightarrow$ Calculated Score

Table 13: Metrics per LLM for project *todo-api*

Model	TC	T	FT	PT	SR	C	M	S
Claude 3.5 Sonnet	$ 0,51	49	3	46	93,9%	85,7%	48,7%	78,1%
GPT 4o	$ 0,28	32	0	32	100,0%	85,7%	44,8%	76,8%
Sabiá 3	$ 0,11	45	0	45	100,0%	80,4%	47,2%	75,8%
LLaMA 3.2 90b	$ 0,02	44	3	41	93,2%	85,7%	50,2%	76,4%
Mistral Large	$ 0,30	40	4	36	90,0%	82,1%	45,2%	72,5%
Gemini 1.5 Pro	$ 0,21	53	6	47	88,7%	85,7%	21,4%	65,3%
Deepseek R1	$ 0,95	34	1	33	97,1%	78,6%	43,5%	73,0%
Qwen 2.5 32b	$ 0,12	47	0	47	100,0%	73,2%	47,8%	73,7%

Table 14: Metrics per LLM for project *restaurants-api*

Model	**TC**	**T**	**FT**	**PT**	**SR**	**C**	**M**	**S**
Claude 3.5 Sonnet	$ 0,60	53	0	53	100,0%	75,8%	32,2%	69,3%
GPT 4o	$ 0,34	31	0	31	100,0%	57,8%	23,9%	60,6%
Sabiá 3	$ 0,12	27	4	23	85,2%	45,3%	19,3%	49,9%
LLaMA 3.2 90b	$ 0,02	37	0	37	100,0%	64,1%	21,9%	62,0%
Mistral Large	$ 0,44	57	1	56	98,2%	64,1%	24,8%	62,4%
Gemini 1.5 Pro	$ 0,27	59	0	59	100,0%	64,1%	26,2%	63,4%
Deepseek R1	$ 0,97	57	0	57	100,0%	75,0%	31,4%	68,8%
Qwen 2.5 32b	$ 0,11	44	0	44	100,0%	68,8%	25,4%	64,7%

Table 15: Metrics per LLM for project *shortener-api*

Model	**TC**	**T**	**FT**	**PT**	**SR**	**C**	**M**	**S**
Claude 3.5 Sonnet	$ 0,25	13	0	13	100,0%	69,0%	43,1%	70,7%
GPT 4o	$ 0,16	11	0	11	100,0%	65,5%	54,6%	73,3%
Sabiá 3	$ 0,05	8	0	8	100,0%	65,5%	35,7%	67,1%
LLaMA 3.2 90b	$ 0,01	15	0	15	100,0%	66,7%	9,8%	58,8%
Mistral Large	$ 0,17	13	0	13	100,0%	66,7%	9,8%	58,8%
Gemini 1.5 Pro	$ 0,09	15	1	14	93,3%	66,7%	9,8%	56,6%
Deepseek R1	$ 0,53	20	3	17	85,0%	64,3%	38,7%	62,7%
Qwen 2.5 32b	$ 0,05	15	2	13	86,7%	67,9%	40,7%	65,1%

Table 16: Metrics per LLM for project *books-api*

Model	**TC**	**T**	**FT**	**PT**	**SR**	**C**	**M**	**S**
Claude 3.5 Sonnet	$ 0,30	25	0	25	100,0%	84,6%	67,0%	83,9%
GPT 4o	$ 0,17	17	0	17	100,0%	84,6%	37,5%	74,0%
Sabiá 3	$ 0,05	11	0	11	100,0%	84,6%	58,0%	80,9%
LLaMA 3.2 90b	$ 0,01	32	0	32	100,0%	80,8%	38,4%	73,1%
Mistral Large	$ 0,18	17	0	17	100,0%	73,1%	32,1%	68,4%
Gemini 1.5 Pro	$ 0,12	32	0	32	100,0%	84,6%	39,3%	74,6%
Deepseek R1	$ 0,52	28	0	28	100,0%	84,6%	60,7%	81,8%
Qwen 2.5 32b	$ 0,05	24	0	24	100,0%	84,6%	39,3%	74,6%

Table 17: Metrics per LLM for project *hotels-api*

Model	**TC**	**T**	**FT**	**PT**	**SR**	**C**	**M**	**S**
Claude 3.5 Sonnet	$ 0,73	58	0	58	100,0%	44,0%	17,9%	54,0%
GPT 4o	$ 0,31	17	2	15	88,2%	17,9%	8,3%	38,1%
Sabiá 3	$ 0,11	23	0	23	100,0%	42,9%	16,8%	53,2%
LLaMA 3.2 90b	$ 0,02	39	0	39	100,0%	38,1%	15,9%	51,3%
Mistral Large	$ 0,49	89	0	89	100,0%	44,0%	12,3%	52,1%
Gemini 1.5 Pro	$ 0,27	62	2	60	96,8%	44,0%	7,5%	49,4%
Deepseek R1	$ 0,99	25	0	25	100,0%	33,3%	10,5%	47,9%
Qwen 2.5 32b	$ 0,11	31	1	30	96,8%	42,9%	10,6%	50,1%

Table 18: Metrics per LLM for project *supermarket-api*

Model	**TC**	**T**	**FT**	**PT**	**SR**	**C**	**M**	**S**
Claude 3.5 Sonnet	$ 0,44	32	0	32	100,0%	71,2%	36,2%	69,1%
GPT 4o	$ 0,24	23	0	23	100,0%	44,2%	28,6%	57,6%
Sabiá 3	$ 0,07	16	0	16	100,0%	67,3%	31,1%	66,1%
LLaMA 3.2 90b	$ 0,02	50	4	46	92,0%	61,5%	31,6%	61,7%
Mistral Large	$ 0,31	44	0	44	100,0%	48,1%	29,1%	59,1%
Gemini 1.5 Pro	$ 0,16	36	0	36	100,0%	75,0%	31,6%	68,9%
Deepseek R1	$ 0,72	38	0	38	100,0%	69,2%	36,7%	68,7%
Qwen 2.5 32b	$ 0,09	47	5	42	89,4%	75,0%	35,7%	66,7%

APPENDIX C – Full List of Failed Tests

This appendix presents the complete catalog of the 39 tests—out of 1.635 total—that failed during execution, representing a failure rate of 2,38%. Each entry includes the project and model that produced the failing test, the number of tests affected, the failure category, and a detailed description of the observed incorrect behavior. Failures are grouped by project for readability. No failures were observed for the *books-api* project across all evaluated LLMs.

Failure categories and their definitions are:

- **Property Length**: the test used a numeric value or a string length that fell outside the allowed range defined in the specification (e.g., a value below the minimum or above the maximum).
- **Misinterpretation**: the test contained logic that was inconsistent with the problem context or misunderstood a business rule of the target API.
- **Authentication**: the test incorrectly applied or omitted authentication in flows that explicitly required or did not require it.
- **Property Requirement**: the test misclassified mandatory and optional request parameters, either omitting required fields or providing unexpected ones.
- **Required Characters**: the test sent a string that did not satisfy character-composition requirements, such as the presence of uppercase letters, digits, or special characters.
- **JSON Deserialization**: the test was unable to correctly deserialize the JSON response when executing assertions.

Table 19: Failed tests for project *todo-api* - part 1 (10/14 failures)

Model	**Qty**	**Category**	**Description**
Mistral Large	3	Authentication	TC031, TC035, TC039 – Attempt to perform an operation (update, delete, and get) without an authentication token. Before performing these operations, the model also tried to create the resource without a token to obtain the entity identifier, which failed with an unauthorized status code, making it impossible to proceed.
Mistral Large	1	Required Characters	TC011 – Attempt to create a user with a password of maximum length. Although the length was correct, the password contained only lowercase letters and did not include the required digit, causing the server to return an error status code.
LLaMA 3.2 90b	1	Property Length	TC026 – Attempt to create a to-do with a title below the minimum acceptable length to verify that the server rejects the request. The minimum length is 1 character and exactly 1 character was sent, which should be valid; the test expected an error but received a success status code.
LLaMA 3.2 90b	1	Property Length	TC014 – Attempt to create a user with a password of exactly the minimum acceptable length to verify success. The minimum is 6 characters and 5 characters were sent, which correctly produces an error; however, the test expected success, causing it to fail.
LLaMA 3.2 90b	1	Required Characters	TC015 – Attempt to create a user with a password of maximum length. The correct length was used, but the password contained only lowercase letters, omitting the required digit and causing the server to return an error.
Gemini 1.5 Pro	1	Property Length	TC006 – Attempt to create a user with a username below the minimum acceptable length to verify that the server rejects the request. The minimum length is 1 character and exactly 1 character was sent, which should be valid; the test expected an error but received a success status code.
Gemini 1.5 Pro	2	Property Length	TC031, TC045 – Attempt to create and update a to-do with a title below the minimum acceptable size to verify that the server rejects the request. The minimum is 1 character and exactly 1 character was sent, which should be valid; the tests expected errors but received success status codes.

Table 20: Failed tests for project *todo-api* - part 2 (4/14 failures)

Model	**Qty**	**Category**	**Description**
Gemini 1.5 Pro	2	Required Characters	TC014, TC015 – Attempt to create a user with passwords at both the minimum and maximum allowed lengths. Although the lengths were correct, the passwords contained only lowercase letters, omitting required characters and causing the server to return error status codes.
Gemini 1.5 Pro	1	Misinterpretation	TC008 – Attempt to create a user with a username of the minimum acceptable length. The correct size was used, but the test reused an email address that had already been created in a previous test case, causing the server to return a conflict error instead of the expected success status code.
Deepseek R1	1	Authentication	TC028 – Attempt to perform a complete CRUD flow. In some operations an authentication token was correctly included, but in others such as GET, the token was omitted, resulting in unauthorized status codes.

Table 21: Failed tests for project *restaurants-api* (5 failures)

Model	**Qty**	**Category**	**Description**
Sabiá 3	2	JSON Deserialization	TC101, TC107 – Attempt to create and query dishes with logically correct test steps and assertions. The tests failed at runtime due to a JSON deserialization error caused by incorrect handling of the response body in the generated assertion code.
Sabiá 3	2	Property Requirement	TC134, TC135 – Attempt to update a restaurant, but the generated requests were missing mandatory properties. The server returned a different status code than expected because the required fields were absent from the request body.
Mistral Large	1	Property Length	TC015 – Attempt to create a dish with a price at the minimum acceptable value. The minimum price is $0.01 and $0.00 was sent, which should produce a validation error. The test expected success, so the returned error caused it to fail.

Table 22: Failed tests for project *shortener-api* (6 failures)

Model	**Qty**	**Category**	**Description**
Gemini 1.5 Pro	1	Property Requirement	TC010 – Attempt to shorten a URL without providing the `qrCode` parameter, expecting the server to return a validation error. However, this parameter is optional according to the specification, and the server processed the request successfully, causing the test to fail.
Deepseek R1	1	Misinterpretation	TC012 – Attempt to shorten a URL by including non-existent parameters in the request, expecting the server to return an error. However, the server silently ignores unknown parameters, causing the request to succeed and the test to fail.
Deepseek R1	1	Property Length	TC016 – Attempt to shorten a URL with a length above the allowed maximum of 2040 characters; 2060 characters were sent. The test expected success but the server correctly returned an error, indicating a boundary misinterpretation.
Deepseek R1	1	Misinterpretation	TC017 – Attempt to retrieve an expired URL after its expiration time. The model applied the required delay at the wrong position in the test—before all operations instead of between the creation and retrieval steps—causing the URL to still be valid at the time of the assertion.
Qwen 2.5 32b	1	Property Length	TC011 – Attempt to shorten a URL with an expiration value below the minimum allowed. The minimum value is 1 and 0 was sent. The test expected success but the server returned a validation error, indicating a boundary misinterpretation.
Qwen 2.5 32b	1	JSON Deserialization	TC013 – Attempt to shorten a URL without a `qrCode` parameter. The test logic was generally correct, but the assertion code failed due to a JSON deserialization error when processing the response body.

Table 23: Failed tests for project *hotels-api* (5 failures)

Model	**Qty**	**Category**	**Description**
GPT 4o	2	Property Requirement	TC008, TC009 – Attempt to use a refresh token to obtain a new authentication token. The generated requests omitted a mandatory property required by the endpoint, causing the server to reject the request with a different status code than expected.
Gemini 1.5 Pro	2	Misinterpretation	TC024, TC030 – Attempt to create and update a country using a name that is already registered, expecting the server to return a duplicate-entry error. However, no such uniqueness constraint exists in the specification or the API logic, so the operations succeeded and the tests failed.
Qwen 2.5 32b	1	Authentication	TC011 – Attempt to query country data without providing an authentication token, expecting the server to return an unauthorized error. However, this endpoint does not require authentication according to the specification, causing the server to return a success response and the test to fail.

Table 24: Failed tests for project *supermarket-api* (9 failures)

Model	**Qty**	**Category**	**Description**
LLaMA 3.2 90b	2	Property Length	TC008, TC016 – Attempt to create and update a category with a name exceeding the maximum allowed length to verify that the server returns a validation error. The maximum is 30 characters and 41 characters were sent. The test expected success but the server correctly returned an error.
LLaMA 3.2 90b	2	Property Length	TC030, TC043 – Attempt to create and update a product with a name exceeding the maximum allowed length to verify that the server returns a validation error. The maximum is 50 characters and 52 characters were sent. The test expected success but the server correctly returned an error.
Qwen 2.5 32b	1	Misinterpretation	TC023 – Attempt to list products filtered by a non-existent `categoryId`, expecting the server to return an error. However, the API correctly returns an empty list when no products match the filter, which is the expected behavior according to the specification; the test incorrectly expected an error response.
Qwen 2.5 32b	1	Misinterpretation	TC046 – Attempt to delete a product using a non-existent identifier, expecting the server to return an error. The specification indicates that the endpoint should return a not-found response in this case, which it did; however, the test assertion logic was inconsistent with this expectation.
Qwen 2.5 32b	2	Property Length	TC009, TC017 – Attempt to create and update a category with a name exceeding the maximum allowed length of 30 characters; 32 characters were sent. The test expected success but the server correctly returned a validation error.
Qwen 2.5 32b	1	Property Length	TC030 – Attempt to create a product with a name above the maximum allowed size of 50 characters; 42 characters were sent, which is within the allowed range. The server returned a success response as expected, but the test incorrectly expected a validation error.

APPENDIX D – Survey Questionnaire

This appendix presents the complete questionnaire used in the practitioners' survey described in Chapter 6. The instrument was distributed anonymously via Google Forms. Submission of the form constituted implicit agreement to the *Termo de Consentimento Livre e Esclarecido (TCLE)*, presented to participants before the first question.

Survey Introduction (shown to respondents)

This survey contributes to a scientific experiment on an approach for generating integration tests using generative artificial intelligence (LLM). The proposed approach, RestTSLLM, uses readily available models with prompt engineering techniques (few-shot and decomposed prompting). The process occurs in two steps: first, from an OpenAPI specification, test scenarios are generated using TSL (Test Specification Language, a structured way to describe test cases); second, from the generated specifications, functional integration tests are produced.

The objective of this research is to evaluate the perception, adoption, and usage intention of this approach by professionals in areas related to the REST API development lifecycle. All collected material is intended exclusively for academic purposes and scientific content production.

The form is designed to be simple and quick, taking approximately 10 minutes. It consists of 3 main blocks: understanding the participant's profile, perception of adoption/usage intention, and suggestions.

The survey is completely anonymous — no personal information will be collected, not even your email address, ensuring your privacy. By completing and submitting the form, you automatically agree to the Free and Informed Consent Form (TCLE).

Block 1 — Participant Profile

Q1.1 What is your current role? *(Single choice)*

- ☐ Software Engineer
- ☐ Test Engineer or QA
- ☐ Tech Lead or Tech Manager
- ☐ Software Architect, Staff Engineer, or Principal Engineer
- ☐ DevOps Engineer or SRE
- ☐ Product Owner / Manager
- ☐ Teacher and/or Researcher with focus on Software Engineering
- ☐ Undergraduate or graduate student in Engineering or Computer Science
- ☐ Other _____

Q1.2 Which of the following apply to you? *(Multiple choice)*

- ☐ Participation in the REST API development lifecycle
- ☐ Practical experience with integration testing or QA
- ☐ Familiarity with OpenAPI, Swagger, Postman, or similar tools
- ☐ Knowledge of or interest in LLMs / AI applied to development

Q1.3 How long have you been working in software engineering? *(Single choice)*

- ☐ Less than 1 year
- ☐ Between 1 and 2 years
- ☐ Between 2 and 5 years
- ☐ Between 5 and 10 years
- ☐ More than 10 years

Q1.4 Which language/stack do you use on a daily basis? *(Multiple choice)*

- ☐ C / C++
- ☐ Java

☐ C#

☐ JavaScript

☐ PHP

☐ Python

☐ Go

☐ Other _____

Q1.5 Do you work with integration tests? *(Yes / No)*

Q1.6 Have you used OpenAPI for API specification or understanding? *(Yes / No)*

Q1.7 Have you used LLMs to assist with technical tasks? *(Yes / No)*

Q1.8 Have you used TSL (Test Specification Language) or a similar criterion to define and specify test cases? *(Yes / No)*

Block 2 — RestTSLLM Approach: Adoption and Usage Intention

Approach Description (shown to respondents before Q2.0)

The RestTSLLM approach proposes, in general terms, guiding an LLM — through examples — to interpret an OpenAPI specification of a REST API and convert it into test cases described in TSL (Test Specification Language). In this approach, the TSL representation is operationalized through yaml files, in which each test case is described in a structured manner. In a subsequent and independent step, the LLM is guided to transform the TSL test cases into functional and executable integration tests, implemented in this experiment in .NET.

As an example, consider the Books API, whose OpenAPI specification describes a REST API for book management, composed of operations to list, create, query, update, and remove books. From this specification, the LLM generates test cases in TSL, which are subsequently converted into executable integration tests.

- Swagger example
- TSL example
- Integration test example

For projects with a large number of endpoints, the approach adopts an endpoint partitioning strategy to make test generation feasible. This strategy is necessary because, in large projects, the test code output may exceed the LLM's context window.

An article was produced describing the experimental evaluation applied and presenting an analysis of the results obtained from applying the approach to six open-source projects. The article was published in the 39th Brazilian Symposium on Software Engineering in 2025 and is available online.

The repository containing the code used for integration with different models, as well as all the results obtained, is available on GitHub.

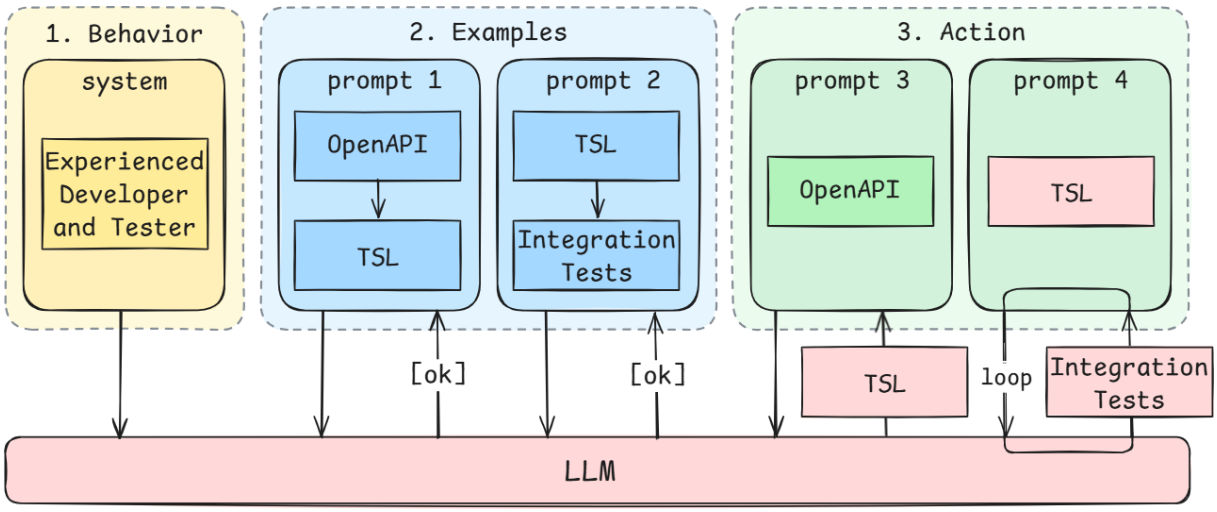

Figure 20: Prompt engineering flow of the RestTSLLM approach.

Q2.0 Based on the description above, select the option that best summarizes the RestTSLLM approach: *(Single choice — eliminatory)*

☐ A strategy that uses LLMs to directly generate integration tests for REST APIs from OpenAPI specifications, without an intermediate step, prioritizing fast production of executable code.

- ☐ A strategy that uses TSL to manually specify REST API test scenarios, allowing developers to subsequently implement integration tests with less dependence on LLMs.
- ☐ A strategy that uses LLMs to enrich OpenAPI specifications with rules, constraints, and examples, so that other automatic test generation tools can use these enhanced specifications as input.
- ☐ A strategy that uses LLMs to generate integration tests for REST APIs from OpenAPI specifications, using TSL as an intermediate representation to structure test scenarios before generating executable code. *(correct answer)*

On a scale of 1 to 5, where 1 corresponds to "Strongly disagree"and 5 corresponds to "Strongly agree,"indicate your degree of agreement with each of the following statements.

Q2.1 The RestTSLLM approach is **useful for automating integration tests**. *(1–5)*

Q2.2 The RestTSLLM approach **can be useful in my day-to-day work**. *(1–5)*

Q2.3 The **separation between "designing test cases" and "generating test code"** facilitates the use of LLMs for testing. *(1–5)*

Q2.4 I would **trust running and using** tests automatically generated by the RestTSLLM approach. *(1–5)*

Q2.5 I **would use** a tool implementing the RestTSLLM approach **if it were available in my team**. *(1–5)*

Q2.6 I believe that integration test generation with RestTSLLM is **applicable in industrial and large-scale scenarios**. *(1–5)*

Q2.7 Which integration modalities would you find most useful for this approach? *(Multiple choice)*

- ☐ IDE plugin
- ☐ Agent or skill (LLM extension)
- ☐ CLI (Command Line Interface)

☐ Web service

☐ CI/CD pipeline

☐ Other: _____

Block 3 — Suggestions

Q3.1 What would stand out most to you — positively or negatively — when using the RestTSLLM approach to support test generation? *(Open-ended, optional)*

Q3.2 What suggestions would you make to make the RestTSLLM approach more useful for your context? *(Open-ended, optional)*

APPENDIX E – Free and Informed Consent Form (TCLE)

This appendix reproduces the *Free and Informed Consent Form* (TCLE — Termo de Consentimento Livre e Esclarecido) used in the acceptance survey described in Chapter 6. The document was approved by the institutional research ethics committee — *Comitê de Ética e Pesquisa (CEP)*, from *Universidade Federal Fluminense (UFF)*, humans committee (Comitê de Humanas) — under registration number *CAAE 89588625.8.0000.8160*, and presented to participants at the beginning of the questionnaire. As the survey was anonymous and administered through Google Forms, submission of the form constituted implicit agreement to the terms described below. The instrument was administered to participants in Portuguese; the version presented here is an English translation, faithful to the original document approved by the CEP.

Free and Informed Consent Form (TCLE)

Universidade Federal Fluminense

Institute of Computing

Survey on the Acceptance of Integration Test Automation Strategies with LLM

Principal Researcher: Thiago da Silva Barradas
Phone: (21) 99329-9143
E-mail: `thbarradas@id.uff.br`
Research Institution: IC/UFF

You are being invited to participate in the research project "Survey on the Acceptance of Integration Test Automation Strategies with LLM," under the responsibility of the researchers of the Institute of Computing of the Universidade Federal Fluminense, Thiago da Silva Barradas and Vânia de Oliveira Neves.

This research is developing an approach for the generation of integration tests using generative artificial intelligence (LLM). The proposed approach, RestTSLLM, employs readily usable native models together with prompt engineering techniques (few-shot and decomposed prompting). The processing occurs in two steps: first, from an OpenAPI specification, test scenarios are generated using TSL; second, from the generated specifications, functional integration tests are generated.

Thus, you will participate in an experiment aimed at evaluating the perception, adoption, and intention to use this approach by professionals in areas related to the REST API development lifecycle.

Your participation will allow you to come into contact with an innovative approach for generating integration tests for REST APIs, broadening your technical and professional experience. For the research, your response will enable progress in the development and validation of the approach, identifying gaps and opportunities to improve its usability and efficiency in real-world environments. For academia, the empirical data will support new scientific publications on the use of LLMs for the automatic generation of integration tests for REST APIs, and will encourage interaction between academia and industry, connecting research with real problems faced by industry. It will also enable, for industry, a practical evaluation of an approach that may optimize the testing process, reducing costs and increasing efficiency in software development.

The research will be conducted anonymously through a Google Forms questionnaire, without storing any information such as name, e-mail, visual information, or any other information that could identify the participant.

During the experiment, if you feel uncomfortable for any reason, you may choose not to respond or even cancel your participation in the experiment at that same moment, without any need for explanation to this researcher or to any other party related to this research. If the experiment is interrupted, all data collected related to it will be completely deleted and will not be included in the data analysis of the research.

In order to mitigate discomforts such as tiredness, annoyance, or other feelings during the completion of the form, it was designed to allow completion in approximately 10 minutes and will be carried out asynchronously.

All material collected in the experiment will be stored on Google's servers, with access restricted to the principal researcher, for a period of 1 (one) year after the conclusion of the research or the defense of the dissertation, whichever occurs first, being deleted after that period. Although Google Forms adopts recognized security measures, there is a residual

risk of breach of confidentiality inherent to cloud storage. This risk is minimized by the fact that the form does not collect any personal identification data, ensuring the anonymity of participants even in the event of any improper access. Thus, the confidentiality and privacy of participants will be maintained. We will not include, under any hypothesis or circumstance, the name or other personal information of any participant, even if the participant has refused or withdrawn from the experiment. All collected material is intended exclusively for academic purposes, for the production of scientific content.

The consolidated results of the research will be made publicly available after its conclusion, through academic publication (a scientific article and/or master's dissertation in the institutional repository of UFF). As the questionnaire is anonymous, feedback to participants will occur exclusively through this public means, without individual communication.

As this research is voluntary, with no cost to the participant, your consent may be withdrawn at any time, without any kind of harm or any other penalty. In addition, this research will not provide any payment to those who participate in the experiment.

Therefore, no expense of the volunteer will be provided or reimbursed. Participation requires only internet access and an electronic device, with an estimated data consumption of less than 1 MB, given the exclusively textual format of the form, equivalent to everyday internet use. To clarify any doubts regarding the procedures, risks, benefits, and other matters related to the research, simply contact the principal researcher.

Below are the details of the Research Ethics Committee that approved this research, so that you may obtain further clarification or information about the study and your participation, should you wish.

Research Ethics Committees (CEPs) are composed of people who work to ensure that all research projects involving human beings are approved in accordance with the ethical standards established by the Ministry of Health. The evaluation by the CEPs takes into account the benefits and risks, seeking to minimize them, and aims to guarantee that participants have access to all the rights assured by regulatory agencies. Thus, the CEPs seek to defend the dignity and interests of participants, encouraging their autonomy and voluntary participation. Find out whether this project was approved by the CEP of this institution. In case of doubts, or if you would like further information, contact the Research Ethics Committee in the Humanities area of the Universidade Federal Fluminense (CEP HUMANAS/UFF). Address: Campus da Praia Vermelha - UFF - Institute of Physics (3rd floor - new tower). Rua Passo da Pátria, 156. Neighborhood: Boa Viagem. Niterói - RJ. E-mail: `eticahumanas.comite@id.uff.br`.

I, _____, declare that I have been informed and agree to be a participant in the research project described above.

_____, _____ of _____ of _____

_____

(signature of participant or legal guardian)

APPENDIX F – Invitation Letter

This appendix reproduces the invitation letter (*Carta Convite*) used to recruit participants for the acceptance survey described in Chapter 6. Recruitment followed a non-probabilistic purposive sampling strategy (Section 6.2.1), and the invitation was distributed through two channels: *individual invitations* sent via LinkedIn direct messages to practitioners matching the target profile, and *collective invitations* posted to technology-focused communities on WhatsApp, Telegram, and Slack. Two wordings were used — one for individual and one for collective invitations. The invitations were administered to participants in Portuguese; the versions presented here are English translations, faithful to the original texts.

Contact and recruitment channels:

- LinkedIn, through a selection of varied professionals matching the defined profile.
- Technology groups on social networks with professionals matching the defined profile, such as WhatsApp, Telegram, and Slack.

For this purpose, two approaches were used: individual invitation and collective invitation.

Individual Invitation

Hi [Name]! How are you?

I am conducting a survey as part of my master's dissertation in Software Engineering. The study evaluates the use of Language Models (LLMs) to automate the generation of integration tests for REST APIs from OpenAPI specifications.

I would like to invite you to answer a brief survey that takes about 10 minutes. The responses are anonymous and will be used exclusively for academic purposes.

Link to the survey

If you are software engineer, QA, test analyst, technology lead, DevOps, or computer science researcher, teacher, or student, with some knowledge of REST API development,

integration testing, familiarity with OpenAPI, Swagger, Postman, or similar tools, and basic knowledge of LLMs/AI applied to development, your opinion will be extremely valuable!

Thank you very much in advance for your collaboration! If you have any questions, I am at your disposal.

Best regards, Thiago Barradas.

Collective Invitation

Hi everyone! How are you?

I am conducting a survey as part of my master's dissertation in Software Engineering. The study evaluates the use of Language Models (LLMs) to automate the generation of integration tests for REST APIs from OpenAPI specifications.

I would like to invite you all to answer a brief survey that takes about 10 minutes. The responses are anonymous and will be used exclusively for academic purposes.

Link to the survey

If you are software engineer, QA, test analyst, technology lead, DevOps, or computer science researcher, teacher, or student, with some knowledge of REST API development, integration testing, familiarity with OpenAPI, Swagger, Postman, or similar tools, and basic knowledge of LLMs/AI applied to development, your opinion will be extremely valuable!

Thank you very much in advance for your collaboration! If you have any questions, I am at your disposal.

Best regards, Thiago Barradas.

APPENDIX G – Open Coding of the Open-Ended Survey Responses

This appendix presents the complete open coding of the two open-ended survey questions (Q3.1 and Q3.2), supporting the qualitative analysis in Section 7.3. Each response is listed once, followed by the code(s) derived from it through open coding (GLASER; STRAUSS, 1967). A single response may yield more than one code; each code is reported with the specific quote (excerpt) that motivated it, its category (Positive, Concern, or Suggestion), and its sub-category. Responses were collected anonymously in Portuguese and translated into English by the author; respondents are identified by an anonymous code (R#). The full machine-readable coding (original Portuguese and English) is available as supplementary material.

Legend: each derived code is shown as **[Category → Sub-category]** *Code* — "quote".

The coding below covers the 50 valid responses to Q3.1 and the 34 to Q3.2 that contained text.

Q3.1 — Salient aspects (positive or negative)

R001. *"Positive point: the possibility of creating tests from the API specification with little human intervention."*

- **[Positive → Productivity gains]** Time/effort reduction — "little human intervention"
- **[Positive → Reuse and standardization]** Generation from the existing OpenAPI — "creating tests from the API specification"

R003. *"Test coverage."*

- **[Positive → Breadth of coverage]** Broad scenario coverage — "Test coverage"

R005. *"Having very high or very low coverage."*

- **[Suggestion → Coverage optimization]** Coverage stability — "Having very high or very low coverage"

R006. *"It would be important for the tests to adapt to behavior changes in the code."*

- **[Suggestion → Test self-adjustment and evolution]** Tests keeping up with code changes — "the tests to adapt to behavior changes in the code"

R014. *"Optimization of time, efficiency, and better final decision-making."*

- **[Positive → Productivity gains]** Time/effort reduction — "Optimization of time, efficiency"

R016. *"The automation in test creation. The productivity gain can be really high."*

- **[Positive → Productivity gains]** Productivity gain in the cycle — "The productivity gain can be really high"

R017. *"The ease of automating test creation and the breadth of tests in systems with many endpoints or even legacy systems."*

- **[Positive → Breadth of coverage]** Broad scenario coverage — "the breadth of tests in systems with many endpoints or even legacy systems"
- **[Positive → Productivity gains]** Automation of test creation — "The ease of automating test creation"

R020. *"If only there were a tool that itself documented the tests in Gherkin."*

- **[Suggestion → Documentation evolution]** Human-readable documentation (Gherkin) — "documented the tests in Gherkin"

R022. *"The time saved in deliveries combined with more robust test coverage."*

- **[Positive → Breadth of coverage]** Broad scenario coverage — "more robust test coverage"
- **[Positive → Productivity gains]** Productivity gain in the cycle — "The time saved in deliveries"

R024. *"Automating test generation with quality and safety is always a huge performance gain for the development cycle, increasing both the quality and the agility of deliveries."*

- **[Positive → Productivity gains]** Productivity gain in the cycle — "a huge performance gain for the development cycle"

R025. *"Having the possibility of creating the yaml directly from the Swagger descriptions and automatically being able to generate the tests based on that is impressive. The negative point, I believe, is the time devoted to reviewing the tests created to ensure they are testing correctly, which is a recurring point in the use of AI in general in the development environment."*

- **[Concern → Trust and human review]** Need for human review — "the time devoted to reviewing the tests created"
- **[Positive → Reuse and standardization]** Generation from the existing OpenAPI — "creating the *yaml* directly from the Swagger descriptions and automatically being able to generate the tests"

R026. *"Positives: time savings, standardization of writing and of practice at the company level. Negatives: tying oneself to restricted and enterprise technologies inhibits learning proficiency and tool updates, a possible learning curve to effectively handle the model, different needs across squads, requiring a more generic model to serve everyone."*

- **[Positive → Productivity gains]** Time/effort reduction — "time savings"
- **[Concern → Limitations of the approach]** Learning curve — "a possible learning curve"
- **[Concern → Limitations of the approach]** Technology lock-in — "tying oneself to restricted and enterprise technologies"
- **[Positive → Reuse and standardization]** Test standardization — "standardization of writing and of practice at the company level"

R028. *"Integration tests are generally more complicated, and anything we can have to make execution easier is very good. In the case of RestTSLLM, I noticed it has a very fluid and familiar direction to adapt to, so I look favorably on its whole proposal to help in development."*

- **[Positive → Adaptable and specialized approach]** Adaptability of the approach — "a very fluid and familiar direction to adapt to"
- **[Positive → Productivity gains]** Productivity gain in the cycle — "anything we can have to make execution easier"

R030. *"It would catch my attention if the generated scenarios and test cases had good variation, going beyond the most basic success and failure tests."*

- **[Positive → Breadth of coverage]** Scenarios beyond the trivial — "good variation, going beyond the most basic success and failure tests"

R031. *"Generation based on the API's own specifications, ensuring at least coverage of all endpoints, validating input and output parameters according to what is actually used."*

- **[Positive → Breadth of coverage]** Coverage of endpoints and parameters — "coverage of all endpoints, validating input and output parameters"
- **[Positive → Reuse and standardization]** Generation from the existing OpenAPI — "Generation based on the API's own specifications"

R032. *"The most positive aspect of the RestTSLLM approach is its automation capability, allowing the LLM to interpret REST API specifications through OpenAPI and convert them into test scenarios."*

- **[Positive → Productivity gains]** Automation of test creation — "its automation capability"
- **[Positive → Reuse and standardization]** Generation from the existing OpenAPI — "interpret REST API specifications through OpenAPI and convert them into test scenarios"

R033. *"In my current development pipeline, I do this process of specifying the scenarios and generating the tests with a dedicated skill/prompt/agent for each interaction, and I*

review the output of the process. I do not have anything more standardized, and this matter of the TSL catches my attention because of that."

- **[Positive → Benefit of the intermediate TSL layer]** Determinism via TSL — "this matter of the TSL catches my attention"
- **[Concern → Trust and human review]** Need for human review — "I review the output of the process"

R038. *"Positively: the strongest point is the use of the TSL as an intermediate layer. Forcing the LLM to structure the test logic before generating the executable code drastically reduces the risk of hallucinations and ensures that the OpenAPI rules are respected. Negatively: the approach becomes very dependent on the quality of the initial OpenAPI specification. If the Swagger/OpenAPI is outdated or poorly documented, the tool will generate incomplete or wrong tests."*

- **[Positive → Benefit of the intermediate TSL layer]** Reduction of hallucinations — "Forcing the LLM to structure the test logic before generating the executable code drastically reduces the risk of hallucinations"
- **[Concern → Specification quality]** Dependency on OpenAPI quality — "very dependent on the quality of the initial OpenAPI specification"
- **[Concern → Specification quality]** Incomplete/outdated specification — "If the Swagger/OpenAPI is outdated or poorly documented"

R040. *"Ability to interpret corner cases."*

- **[Positive → Breadth of coverage]** Scenarios beyond the trivial — "interpret corner cases"

R041. *"It would speed up the delivery of improvements."*

- **[Positive → Productivity gains]** Productivity gain in the cycle — "speed up the delivery of improvements"

R043. *"The possibility of automating the generation of tests from OpenAPI specifications is quite positive. As a point of attention, the quality of the generated tests may vary depending on the level of detail of the specification provided."*

- **[Concern → Specification quality]** Level of specification detail — "the quality of the generated tests may vary depending on the level of detail of the specification"
- **[Positive → Reuse and standardization]** Generation from the existing OpenAPI — "automating the generation of tests from OpenAPI specifications"

R044. *"Helping to generate tests and scenarios that a programmer alone might not be able to identify."*

- **[Positive → Breadth of coverage]** Scenarios beyond the trivial — "scenarios that a programmer alone might not be able to identify"

R047. *"Standardization with related services."*

- **[Positive → Reuse and standardization]** Test standardization — "Standardization with related services"

R048. *"Positively: the possibility of manual review of the result in TSL, as a way to add redundancy to the process of reviewing code, which can increase the number of errors that go unnoticed."*

- **[Positive → Benefit of the intermediate TSL layer]** Auditability and review before code — "manual review of the result in TSL, as a way to add redundancy"

R049. *"Having a more specialized way of generating the tests is a very good idea, since we move away from the more generalist approach that LLMs have."*

- **[Positive → Adaptable and specialized approach]** Specialized approach to test generation — "a more specialized way of generating the tests"

R050. *"Generating the tests covering the most critical cases of the application."*

- **[Positive → Breadth of coverage]** Coverage of critical cases — "covering the most critical cases of the application"

R051. *"Isolation of the LLM and determinism through the TSL."*

- **[Positive → Benefit of the intermediate TSL layer]** Determinism via TSL — "Isolation of the LLM and determinism through the TSL"

R052. *"Depending on the size of the system, generating the specifications and tests could have an exorbitant cost."*

- **[Concern → Potential high cost]** High cost in large systems — "an exorbitant cost"

R054. *"Positively: acceleration in the creation of scenarios and code generation. Negatively: lack of trust in the generation of test scenarios regarding the coverage of critical flows and complex rules."*

- **[Positive → Productivity gains]** Productivity gain in the cycle — "acceleration in the creation of scenarios and code generation"
- **[Concern → Trust and human review]** Lack of trust in generated tests — "lack of trust in the generation of test scenarios regarding the coverage of critical flows and complex rules"

R056. *"The time saved in understanding what the API does, in order to then generate test cases."*

- **[Positive → Productivity gains]** Time/effort reduction — "The time saved in understanding what the API does"

R059. *"The biggest point is that the LLM is not deterministic and tests need to be deterministic. Every LLM-acceleration approach needs a baseline validation to assist. For example, unit tests have coverage and mutation testing to assist; integration has contracts and synthetics."*

- **[Concern → Trust and human review]** Need for a baseline validation — "needs a baseline validation to assist"
- **[Concern → Limitations of the approach]** Non-deterministic LLM vs deterministic tests — "the LLM is not deterministic and tests need to be deterministic"

R060. *"RestTSLLM could analyze the need for mocks and flag them as preconditions of the pre-written tests. I believe that integration can have several levels (integration with the DB, integration between components, between services) and with each new layer of integration it increases the challenge and the need to mock the environment around the SUTs."*

- **[Concern → Low confidence in complex scenarios]** Mock configuration — "with each new layer of integration it increases the challenge and the need to mock the environment around the SUTs"
- **[Concern → Low confidence in complex scenarios]** Mock configuration — "analyze the need for mocks and flag them as preconditions"

R064. *"Ability to correct on its own the errors found."*

- **[Suggestion → Test self-adjustment and evolution]** Self-correction capability — "correct on its own the errors found"

R065. *"What would most catch my attention positively is the ability to reduce the manual effort in creating integration tests from existing OpenAPI specifications, something that today consumes significant time from the team. Negatively, the dependency on the quality of the OpenAPI specification as input can be a limiter; incomplete or poorly documented specifications tend to generate inconsistent test scenarios, requiring human review before execution."*

- **[Positive → Productivity gains]** Time/effort reduction — "reduce the manual effort in creating integration tests from existing OpenAPI specifications"
- **[Concern → Trust and human review]** Need for human review — "requiring human review before execution"
- **[Concern → Specification quality]** Dependency on OpenAPI quality — "the dependency on the quality of the OpenAPI specification as input"
- **[Concern → Specification quality]** Incomplete/outdated specification — "incomplete or poorly documented specifications"

R067. *"I liked the model, but I would be cautious about using it in a large project."*

- **[Concern → Trust and human review]** Lack of trust in generated tests — "I would be cautious about using it in a large project"

R069. *"The main positive point would be the reduction of time and effort in generating the tests. I also consider interesting the strategy of dividing large systems into smaller parts, which helps to work around the LLMs context limitations and reduces the risk of losing important information during test generation. As the main concern, I would highlight the need for human review of the produced results (human-in-the-loop). Although the approach can automate much of the process, I still wonder to what extent it would be possible to trust the generated tests without additional validation. Moreover, it would be important to assess how much human effort is still needed to review and correct the produced artifacts, since this is a widely discussed issue today in the use of LLMs for Software Engineering."*

- **[Positive → Productivity gains]** Time/effort reduction — "the reduction of time and effort in generating the tests"
- **[Concern → Trust and human review]** Lack of trust in generated tests — "to what extent it would be possible to trust the generated tests without additional validation"
- **[Concern → Trust and human review]** Need for human review — "the need for human review of the produced results (human-in-the-loop)"
- **[Positive → Partitioning strategy]** Endpoint partitioning — "dividing large systems into smaller parts"

R070. *"Generating tests in a simpler and faster way, with broad coverage and high assertiveness of the scenarios. However, it is necessary that the test scenarios are well described and that the generated tests go through the approval of a human who understands how the tests work."*

- **[Positive → Breadth of coverage]** Broad scenario coverage — "broad coverage"
- **[Concern → Trust and human review]** Need for human review — "go through the approval of a human"

R076. *"I would need evidence of how much it gets right and wrong in controlled scenarios."*

- **[Concern → Trust and human review]** Lack of trust in generated tests — "evidence of how much it gets right and wrong in controlled scenarios"

R080. *"Self-correction of failing tests."*

- **[Suggestion → Test self-adjustment and evolution]** Self-correction capability — "Self-correction of failing tests"

R082. *"The ability to generate many unit or integration tests, quickly."*

- **[Positive → Productivity gains]** Productivity gain in the cycle — "generate many unit or integration tests, quickly"

R083. *"What most catches my attention positively is the separation between the generation of the test scenarios (TSL) and the executable code, making the process more organized and reusable, which is already a common practice in engineering when thinking about code quality. I also consider very useful the automatic generation of tests from OpenAPI specifications. Negatively, I see as a point of attention the reliability in complex scenarios or with incomplete documentation, requiring human review of the generated tests."*

- **[Concern → Low confidence in complex scenarios]** Complex test flows — "the reliability in complex scenarios"
- **[Positive → Benefit of the intermediate TSL layer]** Decoupling concept from implementation — "the separation between the generation of the test scenarios (TSL) and the executable code, making the process more organized and reusable"
- **[Concern → Trust and human review]** Need for human review — "requiring human review of the generated tests"
- **[Concern → Specification quality]** Incomplete/outdated specification — "with incomplete documentation"

R084. *"I believe it will bring speed in the creation of integration tests and standardization of the scenarios with OpenAPI. In addition, separating the scope of defining the scenario and generating the code greatly helps with maintenance and scaling."*

- **[Positive → Productivity gains]** Productivity gain in the cycle — "speed in the creation of integration tests"
- **[Positive → Benefit of the intermediate TSL layer]** Decoupling concept from implementation — "separating the scope of defining the scenario and generating the code greatly helps with maintenance and scaling"

- **[Positive → Reuse and standardization]** Test standardization — "standardization of the scenarios with OpenAPI"

R086. *"The possibility of specifying the tests in natural language, making possible a more active participation of the business area."*

- **[Suggestion → Documentation evolution]** Natural-language specification — "specifying the tests in natural language, making possible a more active participation of the business area"

R089. *"Positively, what most catches the attention is the use of the intermediate language TSL (in yaml). This solves one of the biggest problems of LLMs, which is the direct generation of complex code with 'hallucinations'."*

- **[Positive → Benefit of the intermediate TSL layer]** Reduction of hallucinations — "the use of the intermediate language TSL (in *yaml*). This solves one of the biggest problems of LLMs, which is the direct generation of complex code with 'hallucinations"'
- **[Positive → Benefit of the intermediate TSL layer]** Reduction of hallucinations — "the direct generation of complex code with 'hallucinations"'

R091. *"My biggest concern is how it handles tests outside the scope of the examples, in code that works with microservices and messaging, being able to handle tests and avoid problems in adjacent services, where we still do not have example tests for these other services."*

- **[Concern → Low confidence in complex scenarios]** Asynchronous APIs and messaging — "how it handles tests outside the scope of the examples, in code that works with microservices and messaging"

R092. *"The positive is that you can focus on each step. For example, first the concept and then the implementation, but I think the challenge is still maintenance. Something that makes sense today can quickly become outdated. We will soon discover that worse than old code and old documentation are old files and configurations to maintain 'new' code."*

- **[Positive → Benefit of the intermediate TSL layer]** Decoupling concept from implementation — "you can focus on each step"

- **[Concern → Specification quality]** Rapid obsolescence — "Something that makes sense today can quickly become outdated"

R093. *"What would most catch my attention positively is the use of TSL as an intermediate step between the OpenAPI specification and the generation of the tests by the LLM. This seems to make the process more structured and less dependent on generic prompts, helping to generate more coherent tests with better coverage."*

- **[Positive → Breadth of coverage]** Broad scenario coverage — "better coverage"
- **[Positive → Breadth of coverage]** More coherent tests — "helping to generate more coherent tests"
- **[Positive → Benefit of the intermediate TSL layer]** Determinism via TSL — "the use of TSL as an intermediate step"

R094. *"The value of this mechanism lies in using the LLM as the single engine for scenario generation and test automation. As the API ecosystem expands organically (a typical behavior in microservice architectures), the manual effort to maintain test coverage becomes unsustainable. The proposed solution mitigates this problem by automating the cycle end to end."*

- **[Positive → Productivity gains]** Automation of test creation — "automating the cycle end to end"
- **[Positive → Productivity gains]** Time/effort reduction — "the manual effort to maintain test coverage becomes unsustainable"

R095. *"The main positive point of the RestTSLLM approach is the use of TSL as an intermediate representation. This architectural decoupling allows the auditing and structural validation of the scenarios before the computational expense of generating the executable code. The strategy of partitioning the endpoints also shows pragmatism by actively mitigating the LLMs context-window limitations. On the other hand, the biggest point of attention is the potential increase in cost and latency due to the multiple inference steps required for the full flow. In addition, the partitioning may hinder the automatic generation of integration tests for business flows that depend on endpoints allocated in distinct groups."*

- **[Concern → Low confidence in complex scenarios]** Complex test flows — "the

partitioning may hinder the automatic generation of integration tests for business flows that depend on endpoints allocated in distinct groups"

- **[Positive → Benefit of the intermediate TSL layer]** Auditability and review before code — "architectural decoupling allows the auditing and structural validation of the scenarios before the computational expense"
- **[Positive → Partitioning strategy]** Endpoint partitioning — "partitioning the endpoints also shows pragmatism"
- **[Concern → Potential high cost]** High cost in large systems — "the potential increase in cost and latency due to the multiple inference steps"

R096. *"What most catches my attention positively is the separation between the generation of the test cases (TSL) and the generation of the executable code. This approach makes the process more transparent, facilitates the review of the generated scenarios, and reduces the direct dependency on the LLM to produce correct code on the first try. I also consider interesting the use of OpenAPI specifications as the main source, since it leverages artifacts that already exist in API development."*

- **[Positive → Benefit of the intermediate TSL layer]** Auditability and review before code — "facilitates the review of the generated scenarios, and reduces the direct dependency on the LLM"
- **[Positive → Benefit of the intermediate TSL layer]** Decoupling concept from implementation — "the separation between the generation of the test cases (TSL) and the generation of the executable code"
- **[Positive → Reuse and standardization]** Generation from the existing OpenAPI — "the use of OpenAPI specifications as the main source, since it leverages artifacts that already exist"

Q3.2 — Suggestions for improvement

R003. *"Classification by priority, tests for critical and non-critical scenarios."*

- **[Suggestion → Coverage optimization]** Prioritization of critical scenarios — "Classification by priority, tests for critical and non-critical scenarios"

R006. *"I thought about computing the "business coverage", not the code coverage, but the coverage of rules and flows covered."*

- **[Suggestion → Coverage optimization]** Business coverage (rules/flows) — "computing the "business coverage", not the code coverage"

R014. *"Make it available in the AIs for free for users who are already accessing OpenAI, so that over time they have easy access to the tool and the dissemination of the tools is faster."*

- **[Suggestion → Delivery and packaging for use]** Integrated availability in AI platforms — "Make it available in the AIs for free for users"

R016. *"Having an integrated test-data generator, perhaps some tool that does seeding."*

- **[Suggestion → Test-data generation]** Test-data generation (fakers/seed) — "an integrated test-data generator, perhaps some tool that does seeding"

R020. *"Avoid multiple prompts and, from the TSL, invoke agents that produce and run the tests."*

- **[Suggestion → Complementary specialized agents]** Agents that run the tests — "invoke agents that produce and run the tests"

R025. *"It might be possible to have a generic prompt that could be used to generate the tests for any API, only needing to change the part that points to the endpoint or group of endpoints to be tested, perhaps even via a command line for a plugin. Something like -> pluginTSL test EndpointX, EndpointY, and this command calls the templated prompt pointing to endpoints X and Y."*

- **[Suggestion → Delivery and packaging for use]** CLI / plugin — "via a command line for a plugin"
- **[Suggestion → Delivery and packaging for use]** Parameterizable generic prompt — "a generic prompt that could be used to generate the tests for any API"

R026. *"Creating tests involving only specs and documentation via black box is complicated; it is often necessary to do a pre-setup of the database or of external integrations so that the flows work properly, like a GET, for example. Integration tests also tend to be very heavy, even considering isolation (avoiding intermittency between test data generated and carried between tests), the infrastructure needed for an end-to-end test starting from an endpoint (database + kafka + schema registry via docker compose is a real scenario in my team). All this makes the tests much slower and costlier than unit tests, which makes them mainly useful in scenarios of "let us test the integration and validate the output, perhaps via snapshot"; but thinking about validating only business rules, this becomes somewhat unfeasible. That is why I see the need for creating specs of test cases, or manual ones, or for the RestTSLLM approach to do a prior white-box analysis of the flow to generate black-box tests adherent to what is expected and with proper setups. It would also be interesting to have an action-plan execution before the actual writing of tests, thinking of something more automated and without human specs. Still on the scenario of which ends the end-to-end integration would test, and I do not know if it has already been considered in future versions, it would be through other types of data input, such as via kafka (stream), pubsub (messaging) or other technologies. This thinking of a direct communication from the client into our cluster or only isolated to the target application (also because bringing up other teams APIs becomes unfeasible in some scenarios where we have a request lifecycle passing through more than 15 systems). Anyway, the approach seems very nice, I hope to see it out there :)"*

- **[Suggestion → Architecture and process evolution]** White-box pre-analysis — "a prior white-box analysis of the flow to generate black-box tests"
- **[Suggestion → Test-data generation]** Database/integration setup — "a pre-setup of the database or of external integrations"
- **[Suggestion → Support for more languages and technologies]** Support for asynchronous inputs (Kafka/PubSub) — "other types of data input, such as via kafka (stream), pubsub (messaging)"

R030. *"An IDE plugin that recognized the language and helped with manual validation and customization. Since it is something focused on automatic code generation, having the possibility of validating what is being generated is essential to me."*

- **[Suggestion → Delivery and packaging for use]** IDE plugin — "An IDE plugin that recognized the language"

- **[Suggestion → Enabling intermediate validation and review]** Manual validation/customization — "helped with manual validation and customization"

R033. *"At the moment, no. It applies quite well to my context."*

- **[Positive → Interest in adoption]** Applicable in day-to-day work — "At the moment, no. It applies quite well to my context"

R038. *"It would be interesting to have a visual interface or a simple checkpoint that allowed the developer to review and adjust the TSL generated in Step 2 before the LLM spends tokens generating the final integration-test code."*

- **[Suggestion → Enabling intermediate validation and review]** TSL review checkpoint before code — "a simple checkpoint that allowed the developer to review and adjust the TSL generated in Step 2 before the LLM spends tokens"

R041. *"It would be necessary to adjust the repository and create good documentation in order to use the LLM."*

- **[Suggestion → Documentation evolution]** Repository documentation — "adjust the repository and create good documentation"

R043. *"It would be interesting to support languages other than .NET, such as JavaScript/TypeScript and Python, to broaden adoption across different team contexts."*

- **[Suggestion → Support for more languages and technologies]** Support for multiple languages — "support languages other than .NET, such as JavaScript/TypeScript and Python"

R050. *"Code coverage can be a challenge; I think it is interesting to be able to point to the target percentage. Or the most critical cases."*

- **[Suggestion → Coverage optimization]** Set a target coverage percentage — "point to the target percentage"
- **[Suggestion → Coverage optimization]** Prioritization of critical scenarios — "Or the most critical cases"

R051. *"Turning the RestTSLLM approach into a self-contained module (autonomous and isolated) would make it scalable and safe in a real production scenario; that is, it could work as a high-level pure function in the ecosystem, where the raw specification goes in and the ready, validated artifacts come out."*

- **[Suggestion → Delivery and packaging for use]** Self-contained module — "self-contained module (autonomous and isolated)"

R054. *"For more critical or specific scenarios, having the customization of the tests and, afterwards, a review of all generated tests, in addition to tracking the fault-coverage reports produced from the CI/CD integration."*

- **[Suggestion → Context customization support]** Test customization — "the customization of the tests"
- **[Suggestion → CI/CD integration and reporting]** Coverage reporting via CI/CD — "the fault-coverage reports produced from the CI/CD integration"
- **[Suggestion → Enabling intermediate validation and review]** Review of generated tests — "a review of all generated tests"

R059. *"The proposal is good for helping in more scenarios. I would look for ways to create a Dashboard to bring together all the testing layers."*

- **[Suggestion → CI/CD integration and reporting]** Coverage dashboards — "create a Dashboard to bring together all the testing layers"

R060. *"No suggestions."*

- **[(no substantive suggestion) → (no substantive suggestion)]** (no substantive suggestion) — "No suggestions"

R062. *"No comments."*

- **[(no substantive suggestion) → (no substantive suggestion)]** (no substantive suggestion) — "No comments"

R064. *"It seems quite useful to me the way it is."*

- **[Positive → Interest in adoption]** Applicable in day-to-day work — "It seems quite useful to me the way it is"

R065. *"It would be useful to support partial or incremental specifications, since not all APIs in our context have complete OpenAPI contracts. In addition, native integration with CI/CD pipelines and with reporting tools such as Allure would ease adoption by the team without the need for manual adaptations. A mechanism for validating the generated TSL before the conversion into code would also reduce rework."*

- **[Suggestion → CI/CD integration and reporting]** CI/CD integration + reporting (Allure) — "native integration with CI/CD pipelines and with reporting tools such as Allure"
- **[Suggestion → Enabling intermediate validation and review]** TSL review checkpoint before code — "A mechanism for validating the generated TSL before the conversion into code"
- **[Suggestion → Support for partial or outdated specifications]** Support for partial/incremental specs — "support partial or incremental specifications"

R067. *"I found it interesting; I would try the approach on small projects and PoCs first to see how it behaves."*

- **[Positive → Interest in adoption]** Would try the approach — "I would try the approach on small projects and PoCs"

R069. *"I suggest considering the integration of evaluator agents (Agent-as-a-Judge) to complement the test generation process. The motivation is that assessing the quality of artifacts produced by LLMs remains an open challenge in Software Engineering. Thus, a specialized agent could perform a preliminary analysis of the generated test cases, identifying possible gaps, inconsistencies, or uncovered scenarios before human validation. This would allow investigating whether part of the effort currently devoted to manual review can be mitigated without compromising the quality of the results."*

- **[Suggestion → Complementary specialized agents]** Evaluator agent (agent-as-a-judge) — "the integration of evaluator agents (Agent-as-a-Judge)"

- **[Suggestion → Enabling intermediate validation and review]** Automatic preliminary analysis — "a preliminary analysis of the generated test cases, identifying possible gaps"

R070. *"Having test scenarios that fit into data-engineering processes, done in a simple and fast way."*

- **[Suggestion → Architecture and process evolution]** Test design for data engineering — "test scenarios that fit into data-engineering processes"

R080. *"Availability via a skill to simplify the prompt."*

- **[Suggestion → Delivery and packaging for use]** Skill — "Availability via a skill to simplify the prompt"

R082. *"Contextualize the 'scenario' or specify the response type."*

- **[Suggestion → Context customization support]** Contextualize scenario/response type — "Contextualize the 'scenario' or specify the response type"

R083. *"It would be interesting to integrate the approach directly into the development flow, via CI/CD, including mechanisms for automatic validation and review of the generated tests. I also consider it important to improve the analysis of incomplete or outdated OpenAPI specifications."*

- **[Suggestion → CI/CD integration and reporting]** CI/CD integration — "integrate the approach directly into the development flow, via CI/CD"
- **[Suggestion → Enabling intermediate validation and review]** Automatic test validation — "mechanisms for automatic validation and review"
- **[Suggestion → Support for partial or outdated specifications]** Improve handling of incomplete/outdated specs — "improve the analysis of incomplete or outdated OpenAPI specifications"

R084. *"Easy ways to validate the generated tests, coverage metrics, and integration with CI/CD pipelines. And also allowing customizations for specific business rules and support for the evolution of the APIs."*

- **[Suggestion → Context customization support]** Customization for business rules — "customizations for specific business rules"
- **[Suggestion → Test self-adjustment and evolution]** Support for API evolution — "support for the evolution of the APIs"
- **[Suggestion → CI/CD integration and reporting]** CI/CD integration + coverage metrics — "validate the generated tests, coverage metrics, and integration with CI/CD pipelines"

R086. *"Not right away, but I would like to learn more deeply about the project."*

- **[Positive → Interest in adoption]** Wants to learn more first — "I would like to learn more deeply about the project"

R089. *"To make the approach more robust, I would suggest including a closed-loop feedback mechanism (self-healing), where the result of the failed test execution could be sent back to the LLM for automatic correction of the generated code. In addition, it would be interesting to support the generation of dynamic test data (fakers) integrated into the TSL, to cover more varied and realistic test scenarios beyond the basics provided in the OpenAPI."*

- **[Suggestion → Test self-adjustment and evolution]** Self-correction capability — "a closed-loop feedback mechanism (self-healing)"
- **[Suggestion → Test-data generation]** Test-data generation (fakers/seed) — "the generation of dynamic test data (fakers)"

R091. *"I think it is hard to think specifically without doing a PoC, but thinking of a checklist it should have: being a not-too-rigid approach, the LLM should also evaluate the examples and the evaluation returned by the intermediate layer; I think the TSL today can be evaluated and understood by AI, and this would open space for less-experienced people to run it too."*

- **[Suggestion → Complementary specialized agents]** Evaluator agent (agent-as-a-judge) — "the LLM should also evaluate the examples"
- **[Positive → Productivity gains]** Faster onboarding of less-experienced staff — "the TSL today can be evaluated and understood by AI, and this would open space for less-experienced people"

R092. *"Validation of the apispec contracts in GitHub Actions, code templates, etc."*

- **[Suggestion → CI/CD integration and reporting]** GitHub Actions integration — "code templates"
- **[Suggestion → CI/CD integration and reporting]** Contract validation in CI (GitHub Actions) — "Validation of the apispec contracts in GitHub Actions"

R094. *"Since the use of LLMs still incurs a high cost, it would be excellent if the library were able to generate the specifications and tests incrementally, restricting itself strictly to the code snippets changed in the commit."*

- **[Suggestion → Test self-adjustment and evolution]** Incremental generation (only changed code) — "generate the specifications and tests incrementally, restricting itself strictly to the code snippets changed in the commit"

R095. *"To optimize the RestTSLLM approach, I suggest aligning the partitioning of endpoints with the Vertical Slicing architecture, ensuring that the tests reflect cohesive business flows. In addition, the generation of the TSL could be integrated into Test-Driven Development (TDD) cycles, creating scenarios that guide the implementation of APIs from the initial failing phase."*

- **[Suggestion → Architecture and process evolution]** Integration with TDD — "the generation of the TSL could be integrated into Test-Driven Development (TDD) cycles"
- **[Suggestion → Architecture and process evolution]** Partitioning with vertical slicing — "aligning the partitioning of endpoints with the Vertical Slicing architecture"

R096. *"I would suggest including configuration mechanisms to adapt the tests to the project context, such as authentication, test data, execution environment, and naming conventions used by the team. It would also be useful to allow the user to review and edit the TSL cases before generating the code, ensuring greater control over the scenarios."*

- **[Suggestion → Context customization support]** Configuration (auth, data, environment, naming) — "configuration mechanisms to adapt the tests to the project context, such as authentication, test data, execution environment, and naming conventions"

- **[Suggestion → Enabling intermediate validation and review]** TSL review checkpoint before code — "allow the user to review and edit the TSL cases before generating the code"