Combining TSL and LLM to Automate REST API Testing: A Comparative Study

Thiago Barradas
Universidade Federal Fluminense
Niterói, RJ, Brazil
thbarradas@id.uff.br

Aline Paes
Universidade Federal Fluminense
Niterói, RJ, Brazil
alinepaes@ic.uff.br

Vânia de Oliveira Neves
Universidade Federal Fluminense
Niterói, RJ, Brazil
vania@ic.uff.br

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 specifications. 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.

KEYWORDS

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

1 Introduction

Software testing is an essential component of the system development lifecycle, playing a crucial role in ensuring the quality and reliability of systems [3, 23, 49]. 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 [3, 23, 49].

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 [31, 35, 49]. Exhaustively testing all possible inputs is unfeasible, especially in systems that require high reliability, which consumes a significant portion of the development effort [49]. Additionally, manual test execution can result in incomplete scenarios and difficulty in detecting subtle failures [49], leading to high costs and limited coverage [23].

Several studies have contributed to advancing software testing techniques, particularly by addressing challenges such as test input generation and the complexities inherent in real-world, modern systems [38]. These challenges were highlighted in the 2014 study by Orso et al. [38] and remain open problems nowadays. 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.

However, this process can be significantly accelerated by the recent emergence of several studies exploring the use of Large Language Models (LLMs) to enhance software testing techniques[9]. 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 [41]. 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 [9, 41].

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 [28]. 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 [4, 28].

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 [16, 35, 44, 45]. Unlike unit tests, which target isolated functionalities, integration tests focus on interactions between services, modules, or systems, ensuring the correct exchange of information, such as inputs, responses, and communication flows. According to Newman [35], failing to ensure proper integration between components can result in critical communication issues, such as inadequate responses. Corradini et al. [11] further emphasize that for REST APIs, validating system behavior in real-world scenarios depends heavily on integration tests.

Recent studies highlight the increasing attention to REST API testing, with emphasis on automation techniques, schema-based fuzzing, and test coverage metrics [17]. Golmohammadi et al. [17] identified black-box 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 [11, 36, 48] 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 [17, 36, 48].

A recent survey by Wang et al. [53] 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[1]. Despite the growing interest in applying LLMs to software testing, Wang et al. [53] 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. [53], the study by Alshahwan et al. [2] 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 techniques such as LLMs. These observations reinforce the need for developing approaches that enhance the robustness and effectiveness of integration testing for REST APIs.

In this context, 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 [24], 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 and edge cases, including validation failures. For this purpose, we propose the use of Test Specification Language (TSL) [39], 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 (RQ):

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

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

To answer these questions, this study adopts prompt engineering with the *few-shot* and *decomposed prompting* techniques [24], 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) [32]. 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 proposed research questions.

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 the study's expectations, proving to be promising tools for this purpose. These tools enabled us to generate more efficient 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.

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.

The data and code supporting the conclusions of this study are publicly available on GitHub [6].

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.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 presents more details about our approach, RestTSLLM, and how it can be replicated. Section 4 describes the methodology used, including details of the selected technologies and tools, the prompts utilized, and the evaluation

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

and comparison metrics used in this study. Section 5 delves into analyzing the results obtained in the experiment. Section 6 discusses threats to the validity and limitations of the study regarding its applicability. Finally, Section 7 presents the conclusions, summarizing the main contributions and outlining possible future work.

2 Related works

This section reviews relevant studies that explore the use of Large Language Models (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 our study.

There is a growing interest in using LLMs across various areas of software engineering, including requirement engineering, code implementation, testing, maintenance, and deployment [13, 20]. Fan et al. [13] 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 et al. [7] 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, it presents limitations in terms of scalability and accuracy, especially in more complex or ill-defined tasks such as integration testing. Although the approach analyzes various development lifecycle phases, it provides limited attention to integration testing and complex testing scenarios.

In comparative performance studies, Mendoza et al. [31] 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 BDD scenarios. The study highlighted that, although some tools struggled to understand certain contexts, GPT 4 and Gemini performed better in generating test data consistent with BDD specifications, outperforming GPT 3.5 and Copilot. The analysis reinforces the importance of adapting LLM usage to improve testing efficiency, particularly in more complex and dynamic scenarios.

The work by Ouédraogo et al. [40] 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, it 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 more complex tasks like integration testing—it highlights the need to evolve test generation methods and refine prompting strategies to improve test coverage and clarity, especially when comparing LLMs to traditional tools.

Kim et al. [26] conducted an experiment on using LLMs to improve REST APIs testing and proposed the RESTGPT approach. The paper mentions that traditional test generation tools for REST APIs typically use an OpenAPI specification as input. However, the tools cannot understand the insights provided due to the lack of rules exposed in a machine-readable way. To address this problem, RESTGPT proposes using an LLM to enrich the information provided in the specification. It takes an OpenAPI file as input and generates constraints, rules, and examples, returning a more detailed and machine-readable OpenAPI specification. This enhanced output can then be used by other automated test generation tools as input. This method showed significant improvements in the accuracy of correctly interpreting parameter descriptions and generating valid values more accurately and contextually than traditional tools such as NLP2REST and ARTE. The study reports notable gains in rule extraction accuracy. However, this work does not aim to generate automated REST API tests, but rather to enrich the raw material used as input for this task.

Expanding upon this line of research, Kim et al. [25] proposed LlamaRestTest, a black-box testing approach based on fine-tuned and quantized LLaMA 3 models to improve input generation and detection of parameter dependencies in REST API testing. Unlike RESTGPT, which statically enriches OpenAPI specifications, LlamaRestTest dynamically adapts test inputs based on server feedback using reinforcement learning. It employs two specialized models: one for identifying dependencies (LlamaREST-IPD) and another for generating valid inputs (LlamaREST-EX). The study shows that small, optimized models can outperform larger ones and existing tools in coverage and fault detection. However, it focuses on dynamic test execution rather than generating reusable test artifacts, making each run unique and harder to reproduce or version.

In contrast to LlamaRestTest, our approach uses a simpler and more accessible workflow that generates reusable test artifacts covering both success and client error scenarios. We use general-purpose LLMs, without the need for prior training, to generate REST API tests only based on examples provided through prompt engineering. We explore the direct use of LLMs in their default state to produce executable tests compatible with traditional tools, promoting easier adoption and integration into developers' daily workflows.

In addition to the previously mentioned approaches that leverage LLMs, studies such as those by Golmohammadi et al. [17], Viglianisi et al. [52], and Corradini et al. [10] analyze and explore the automated generation of REST API tests from OpenAPI specifications using traditional black-box tools, including RESTTestGen, RESTler, bBOXRT, and RESTest. While these tools have shown satisfactory coverage and fault detection results, they still face challenges in balancing these two metrics. It is worth noting that such studies do not explore the use of LLMs to optimize outcomes or to enhance the interpretation of natural language rules present in the specifications—limiting their potential for understanding complex business logic.

Despite the increasing number of studies on using LLMs for test generation, which demonstrate their potential, there remains a lack of research focused on generating and evaluating REST API

tests. Most approaches concentrate on unit testing, with a limited focus on integration tests, and the few that exist for integration testing do not have an approach capable of solving most problems efficiently. In this context, we identify an opportunity to expand research and fill this gap regarding the automated generation of REST API tests, providing a new approach to automating them, and enabling comparative evaluation of effectiveness among existing LLMs, providing a broader understanding of how LLMs can be used to automate the production of integration tests for REST APIs.

3 Our approach

RestTSLLM is an approach designed to address the challenges of REST API testing, particularly in generating scenarios, input data, and executable tests. The method 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. 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 section aims to detail our approach, which is visually summarized in Figure 1.

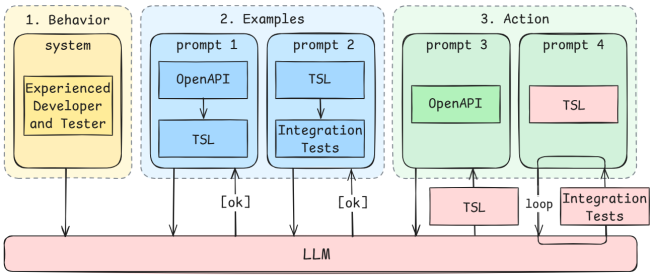

Figure 1: Overview of RestTSLLM approach

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 1 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*.

In the first stage, **Behavior**, the LLM is instructed, through the *system* 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 [33, 34, 56].

In addition to acting as an experienced developer and tester, the model was instructed to operate as if it had a strong knowledge of the target technology used for test generation, of rule extraction from OpenAPI and conversion to TSL using the Category-Partition method [39], 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 was configured to return only the requested code, without textual explanations or justifications.

The second stage, **Examples**, consists of two steps that apply the *few-shot* and *decomposed prompting* techniques [24] 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 *yml*. 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 [39]. This stage teaches the LLM how to interpret OpenAPI 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 [24].

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.

Listing 1 and Listing 2 illustrate the outputs generated by the LLM in response to prompts 3 and 4, respectively. The first block shows a test case written in 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 2 presents the corresponding integration test code generated from the TSL 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 the test generation, although minor adjustments may be applied to ensure repeatability. In the example show in Listing 2, 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 system under test, as taught in the examples presented to LLM.

```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 1: Sample of TSL generated by LLM

```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 2: Sample of integration test generated by LLM from previous TSL

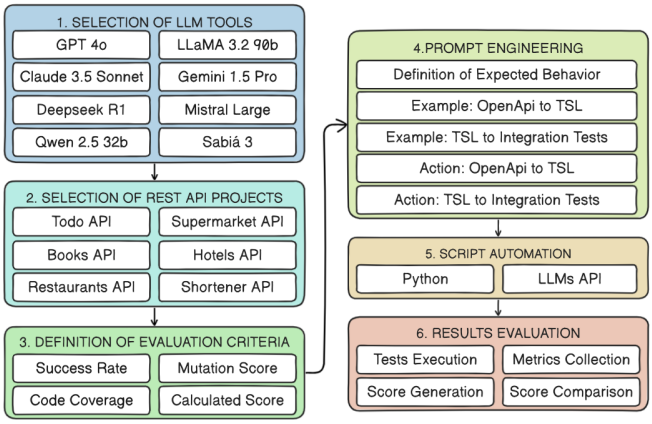

Figure 2: Experimental methodology flow

4 Experimental methodology

This section describes the methodology used to conduct the experiment. Through these steps, we aim to validate the proposed approach by evaluating the effectiveness of the selected LLMs to determine which one best aligns with our expectations for integration test generation. Figure 2 illustrates the methodological steps. Each step is summarized below, with further details presented in the following subsections.

The first step, **Selection of LLM tools**, involves 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 (Subsection 4.1). 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 (Subsection 4.2).

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 select the success rate, coverage, and mutation score to compute a calculated score from these metrics and evaluate the performance of LLMs (Subsection 4.3). The fourth step, **Prompt Engineering**, expands on the experiment details in creating prompts used to guide the LLMs in RestTSLLM approach (Subsection 4.4). The fifth step, **Script Automation**, refers to the development of an automated script for integrating with the LLMs to process the prompts, and their results (Subsection 4.5). Finally, the sixth step, **Results Evaluation**, describes the evaluation of the integration tests generated by the LLMs (Subsection 4.6).

4.1 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, extracted from publicly available online benchmarks [8, 21, 29, 37, 43, 50, 51], and their use in previous studies [5, 19, 42, 55]. Additionally, only models with publicly available documentation, and the ability to be used both via API, and user interface were considered.

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 the Brazilian model Sabiá 3 (Maritaca) [1], developed in the authors' home country, to support analyses in the local language, contributing to technological advancement, and highlight Brazilian research 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.

4.2 Selection of REST API projects

The criteria used for selecting REST API projects for this experiment include: 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 authors, and widely adopted in the industry for REST API development [14].

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. These projects and the evaluation criteria are presented in Table 2. Most had an MIT license[2], allowing free use, while formal permission was obtained for the *hotels-api* project[3]. We also included a *CLOC*[4] column to indicate the size of each project based on the number of lines of C# code.

4.3 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 [22].

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 system under test [40].

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 [57].

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[22, 57].

To compute a *Calculated Score*, we applied the TOPSIS technique (Technique for Order Preference by Similarity to Ideal Solution [30]), derived from the MCDM (Multi-Criteria Decision Making) method[30]. TOPSIS aims to solve decision-making problems that involve multiple conflicting criteria in their analysis. These methods help select the best alternative in situations where different criteria, sometimes conflicting, must be considered simultaneously.

The formula used to calculate the *Calculated Score*, denotated 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*). These metrics do not require normalization, as they are already expressed within a standard range between 0 and 1,

[2] https://opensource.org/license/mit

[3] Authorization formally requested: https://github.com/uffsoftwaretesting/RestTSLLM/tree/main/pdfs/use_permission_hotel_api_.pdf

[4] CLOC stands for "Count Lines Of Code". We used the project available at https://github.com/AlDanial/cloc to count only C# lines.

Table 2: Selected Projects

Name	License	.NET Version	Endpoints	Dependencies	Stars	Forks	Stars + Forks	CLOC
todo-api [15]	MIT	7.0	7	SQLite	2.964	441	3.405	1.576
supermarket-api [18]	MIT	8.0	8	SQLite	484	166	550	977
books-api [46]	MIT	8.0	5	Postgre SQL, Redis	145	92	237	809
hotels-api [54]	Requested	7.0	14	SQL Server	76	42	118	2.843
restaurants-api [27]	MIT	8.0	10	SQL Server, Azure Storage	64	39	103	1.804
shortener-api [12]	MIT	9.0	3	MongoDB, Redis	78	22	100	1.055

represented as percentages. Finally, we applied a weight of $33.3\overline{3}\%$, denotated by w, to each metric to balance and proportionally reflect their relative importance in the calculated score regarding the effectiveness of the generated tests.

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 first 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 efficient.

4.4 Prompt engineering

In addition to the general aspects of the RestTSLLM approach previously explained in Section 3, we will explain in this section the complementary peculiarities of applying the approach in our experimental methodology.

As the target technology for generating REST API integration tests, we chose .NET with xUnit [32], given the authors' 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 [14]. 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 prompt content 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 authors' home country, aiming to support analyses focused on the local language, contribute to technological development, and highlight Brazilian research within the scientific community.

The generic prompts used in this study are available on GitHub[5].

4.5 Script automation

Studies involving LLMs commonly apply prompt engineering in their experiments using the interface provided by the model developer [53]. However, this approach can become complex when executing the same sequence of prompts and collecting their results across a larger set of 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.

To overcome this, we developed a Python script that automates the entire process: loading prompt files, handling model configurations, executing each step while maintaining conversational context, and saving the outputs and metrics. 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.

The source code of the automated script is available on GitHub[6].

4.6 Results evaluation

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[7].

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 Subsection 4.3.

To obtain the success rate and coverage results, we ran the tests using the Visual Studio 2022 tool [32]. We used the branch coverage metric previously explained in Subsection 4.3.

To obtain the mutation score we use the Stryker.NET tool [47] 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

[5]https://github.com/uffsoftwaretesting/RestTSLLM/tree/main/general-files/default-prompts/files.md
[6]https://github.com/uffsoftwaretesting/RestTSLLM/tree/main/llm-processor/README.md
[7]https://github.com/uffsoftwaretesting/RestTSLLM/tree/main/projects/files.md

string, math, and linq [47]. 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 [47].

The metrics collected from the test executions, and the evidence of these executions are available in a public PDF file on GitHub[8] due to space restrictions in this document.

5 Results and discussion

This section 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 combination of success rate, coverage, and mutation score using the TOPSIS technique. The aggregated result, ordered from highest to lowest, are presented in Table 3. The detailed results by LLM and project are available in a public PDF file on our GitHub repository[8]. 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, we use a single letter 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
- $TC \rightarrow$ Total Cost (USD)

Table 3: 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

5.1 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—proved to be satisfactory. All LLMs managed to preserve the expected structure and generate compilable code for integration test execution.

[8]https://github.com/uffsoftwaretesting/RestTSLLM/tree/main/pdfs/all_results.pdf

The content generated was read and manually reviewed. All results were satisfactory in terms of readability, clarity, and adherence to the test patterns presented in the prompts. 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. 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[7].

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 5.3. This indicates that LLMs are capable of producing tests that not only execute successfully but also explore the system under 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.

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

To determine the most efficient LLMs, we computed a final score (average calculated score) for each model, based on a weighted combination of success rate, coverage, and mutation score using the TOPSIS technique. The results, ordered from highest to lowest, are presented in Table 3.

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 efficient, ranking first in all metrics and being the only model that produced no failed tests (Table 5). We analyze however that the models 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 4.

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

Table 4: 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%

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 3, 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.

5.3 Analysis of test failures

Out of the 1.635 tests generated, 39 failed to execute correctly, representing 2,38%. Table 5 details the number of failed tests per 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 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 5: 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%

6 Threats to validity and limitations

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.

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.

OpenAPI Dependency: Our approach depends on the availability and quality of OpenAPI specifications to drive test generation. Although OpenAPI 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.

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 Conclusion

This approach - RestTSLLM - proposed the use of TSL and LLM to automate the generation of integration tests for REST APIs from its OpenAPI specification, and its experimental evaluation assessed its effectiveness. A comparative evaluation of tools was conducted, defining specific prompts, and clear criteria for metric analysis, including success rate, coverage, and mutation score.

The experimental results demonstrated that all evaluated LLMs were capable of generating integration tests with satisfactory quality, based on both subjective and objective evaluation criteria. Notably, 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. It achieved the highest average calculated score, ranked first in all individual metrics, and was the only model that produced no faulty tests. These results reinforce the feasibility of using LLMs to automate integration testing for REST APIs and highlight the potential of prompt-based strategies combined with intermediate TSL generation to guide LLMs more effectively.

In summary, this study 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.

Future work includes addressing the limitations discussed throughout this study, exploring ways to strengthen and broaden the applicability of the proposed approach. First, we plan to conduct additional experiments to validate the generalizability of the method. This includes testing with other emerging LLM tools, applying the approach to more complex architectures (such as event-driven and microservices systems), and using large-scale REST APIs with pre-existing test suites to enable quantitative comparisons. We also aim to evaluate its applicability with different technologies beyond .NET and xUnit.

Second, we intend to enhance the methodology by developing new features that improve automation and usability. This includes automatic handling and correction of errors through reprompting techniques, combining multiple generated tests to optimize coverage and fault detection, implementing IDE plugins to better integrate the approach into developers' workflows, and supporting test maintenance and evolution throughout the software lifecycle, enabling incremental updates, improving support for outdated or incomplete API specifications, and incorporating alternative input formats beyond OpenAPI.

In addition, we plan to investigate the use of additional languages in prompts and examples—beyond Portuguese and English—to evaluate multilingual performance. We will also compare the time and effort of manual test creation versus LLM-assisted generation and study ways to mitigate LLM token limitations in scenarios involving long prompts.

Finally, our approach not only aligns with key stages of the software testing lifecycle for REST APIs—particularly test case preparation and implementation—but also demonstrates its potential to enhance and scale traditional testing practices. These results reinforce the feasibility and relevance of adopting LLMs as a viable strategy for advancing test automation in real-world scenarios.

ARTIFACT AVAILABILITY

The authors declare that the research artifacts supporting the findings of this study are accessible at https://doi.org/10.6084/m9.figshare.29525768.v4. The artifacts can also be found on GitHub, accessible at https://github.com/uffsoftwaretesting/RestTSLLM.

ACKNOWLEDGMENTS

This paper has been supported by CNPq - *National Council for Scientific and Technological Development* (grants 420025/2023-5 and 307088/2023-5), FAPERJ - *Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro*, processes SEI-260003/002930/2024, SEI-260003/000614/2023, and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

REFERENCES

[1] Hugo Abonizio, Thales Sales Almeida, Thiago Laitz, Roseval Malaquias Junior, Giovana Kerche Bonás, Rodrigo Nogueira, and Ramon Pires. 2024. Sabiá-3 Technical Report. https://arxiv.org/abs/2410.12049

[2] Nadia Alshahwan, Mark Harman, and Alexandru Marginean. 2023. Software Testing Research Challenges: An Industrial Perspective. In *2023 IEEE Conference on Software Testing, Verification and Validation*. https://doi.org/10.1109/ICST57152.2023.00008

[3] Abhineet Anand and Azeem Uddin. 2019. Importance of software testing in the process of software development. *International Journal for Scientfic Research and Development (IJSRD)* (2019). https://www.researchgate.net/publication/331223692

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. REST-ler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE Press. https://doi.org/10.1109/ICSE.2019.00083

[5] Omer Aydin, Enis Karaarslan, Fatih Safa Erenay, and Nebojsa Bacanin. 2025. Generative AI in Academic Writing: A Comparison of DeepSeek, Qwen, ChatGPT, Gemini, Llama, Mistral, and Gemma. https://arxiv.org/abs/2503.04765

[6] Thiago Barradas. 2025. Integration Test Generation - LLM Efficiency - Repository. https://github.com/uffsoftwaretesting/RestTSLLM Accessed on: June 26, 2025.

[7] Lenz Belzner, Thomas Gabor, and Martin Wirsing. 2023. Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study. In *AISoLA*. https://doi.org/10.1007/978-3-031-46002-9_23

[8] BotPress. 2025. Discover models by Popularity. https://botpress.com/llm-ranking Accessed on: January 25, 2025.

[9] Mohamed Boukhlif, Nassim Kharmoum, and Mohamed Hanine. 2024. LLMS for intelligent software testing: a comparative study. In *Proceedings of the 7th International Conference on Networking, Intelligent Systems and Security*. https://dl.acm.org/doi/10.1145/3659677.3659749

[10] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical Comparison of Black-box Test Case Generation Tools for RESTful APIs. https://arxiv.org/abs/2108.08196

[11] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato. 2021. Restats: A Test Coverage Tool for RESTful APIs. In *37th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. https://arxiv.org/abs/2108.08209

[12] Hasan Erdal. 2025. Shortener API - Repository. https://github.com/Filiphasan/dotnet-minify-url Accessed on: March 24, 2025.

[13] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. 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)*. https://arxiv.org/abs/2310.03533

[14] Fortune Business Insights. 2023. *Dot Net Development Service Market Size, Share, Growth.* https://www.fortunebusinessinsights.com/dot-net-development-service-market-111361 Accessed: January 12, 2025.

[15] David Fowler. 2025. Todo API - Repository. https://github.com/davidfowl/TodoApp Accessed on: March 24, 2025.

[16] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2022. Testing RESTful APIs: A Survey. https://arxiv.org/abs/2212.14604

[17] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* (2023). https://doi.org/10.1145/3617175

[18] Evandro Gomes. 2025. Supermarket API - Repository. https://github.com/evgomes/supermarket-api Accessed on: March 24, 2025.

[19] Desta Haileselassie Hagos, Rick Battle, and Danda B. Rawat. 2024. Recent Advances in Generative AI and Large Language Models: Current Status, Challenges, and Perspectives. https://arxiv.org/abs/2407.14962

[20] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2024).

[21] This is Definition. 2025. What's the most popular LLM? https://www.thisisdefinition.com/insights/most-popular-llm Accessed on: January 20, 2025.

[22] Kush Jain, Goutamkumar Tulajappa Kalburgi, Claire Le Goues, and Alex Groce. 2023. Mind the Gap: The Difference Between Coverage and Mutation Score Can Guide Testing Efforts. https://arxiv.org/abs/2309.02395

[23] Tanu Jindal. 2016. Importance of Testing in SDLC. *International Journal of Engineering and Applied Computer Science (IJEACS)* (2016). https://www.researchgate.net/publication/312041152

[24] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed Prompting: A Modular Approach for Solving Complex Tasks. https://arxiv.org/abs/2210.02406

[25] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2025. LlamaRestTest: Effective REST API Testing with Small Language Models. https://arxiv.org/abs/2501.08598

[26] Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. 2024. Leveraging Large Language Models to Improve REST API Testing. (2024). https://doi.org/10.1145/3639476.3639769

[27] Jakub Kozera. 2025. Restaurants API - Repository. https://github.com/jakubkozera/Restaurants Accessed on: March 24, 2025.

[28] Alexander Lercher. 2024. Managing API Evolution in Microservice Architecture. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. ACM. https://doi.org/10.1145/3639478.3639800

[29] LiveBench. 2024. LiveBench - Leaderboard. https://livebench.ai/#/ Accessed on: January 25, 2025.

[30] Mitra Madanchian and Hamed Taherdoost. 2023. A comprehensive guide to the TOPSIS method for multi-criteria decision making. *Madanchian M, Taherdoost H. A comprehensive guide to the TOPSIS method for multi-criteria decision making. Sustainable Social Development* (2023).

[31] Isela Mendoza, Fernando Silva Filho, Gustavo Medeiros, Aline Paes, and Vânia Neves. 2024. 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, Porto Alegre, RS, Brasil. https://sol.sbc.org.br/index.php/sbes/article/view/30369

[32] Microsoft. 2025. *Integration tests in ASP.NET Core.* https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests Accessed: April 06, 2025.

[33] Norman Mu, Jonathan Lu, Michael Lavery, and David Wagner. 2025. A Closer Look at System Prompt Robustness. https://arxiv.org/abs/2502.12197

[34] Nebuly. 2024. *LLM System Prompt vs. User Prompt.* https://www.nebuly.com/blog/llm-system-prompt-vs-user-prompt Accessed: April 07, 2025.

[35] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.

[36] Mitchell Olsthoorn. 2022. More Effective Test Case Generation with Multiple Tribes of AI. In *Proceedings of the 44th International Conference on Software Engineering (ICSE) - Doctoral Symposium*. ACM. https://doi.org/10.1145/3510454.3517066

[37] OpenRouter. 2025. LLM Rankings - Programming. https://openrouter.ai/rankings/programming?view=month Accessed on: January 25, 2025.

[38] Alessandro Orso and Gregg Rothermel. 2014. Software testing: a research travelogue (2000-2014). In *Future of Software Engineering Proceedings*. Georgia Institute of Technology. https://dl.acm.org/doi/10.1145/2593882.2593885

[39] Thomas J. Ostrand and Marc J. Balcer. 1988. The category-partition method for specifying and generating fuctional tests. *Commun. ACM* (1988). https://dl.acm.org/doi/10.1145/62959.62964

[40] Wendkûuni C. Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. 2024. Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation. https://arxiv.org/abs/2407.00225

[41] Mauro Pezzè, Matteo Ciniselli, Luca Grazia, Niccolò Puccinelli, and Ketai Qiu. 2024. The Trailer of the ACM 2030 Roadmap for Software Engineering. https://www.inf.usi.ch/faculty/pezze/media/SE2030SENreport.pdf [Online; accessed 1-Apr-2025].

[42] Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2024. How Efficient is LLM-Generated Code? A Rigorous High-Standard Benchmark. https://arxiv.org/abs/2406.06647

[43] Shakudo. 2025. Top 9 Large Language Models as of January 2025. https://www.shakudo.io/blog/top-9-large-language-models Accessed on: January 26, 2025.

[44] Ian Sommerville. 2010. *Software Engineering* (9 ed.). Addison-Wesley, Harlow, England.

[45] Stelios Sotiriadis, Andrus Lehmets, Euripides G. M. Petrakis, and Nik Bessis. 2017. Unit and Integration Testing of Modular Cloud Services. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. https://doi.org/10.1109/AINA.2017.57

[46] Poorna Soysa. 2025. Books API - Repository. https://github.com/poorna-soysa/books-api-docker-compose-postgresql-redis Accessed on: March 24, 2025.

[47] Stryker. 2025. *Stryker .NET - Configuration.* https://stryker-mutator.io/docs/stryker-net/configuration/ Accessed: April 09, 2025.

[48] Martin Tappler, Andrea Pferscher, Bernhard K. Aichernig, and Bettina Könighofer. 2024. Learning and Repair of Deep Reinforcement Learning Policies from Fuzz-Testing Data. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. ACM. https://doi.org/10.1145/3597503.3623311

[49] Maneela Tuteja, Gaurav Dubey, et al. 2012. 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)* (2012). https://www.ijsce.org/portfolio-item/c0761062312/

[50] Unite.AI. 2025. Best Large Language Models (LLMs) in 2025. https://www.unite.ai/best-large-language-models-llms/ Accessed on: January 20, 2025.

[51] Vellum. 2025. LLM Leaderboard - Model Comparison. https://www.vellum.ai/llm-leaderboard Accessed on: January 26, 2025.

[52] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. https://doi.org/10.1109/ICST46399.2020.00024

[53] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. https://arxiv.org/abs/2307.07221

[54] Trevoir Williams. 2025. Hotels API - Repository. https://github.com/trevoirwilliams/HotelListing.API.NET Accessed on: March 24, 2025.

[55] Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top Leaderboard Ranking = Top Coding Proficiency, Always? EvoEval: Evolving Coding Benchmarks via LLM. https://arxiv.org/abs/2403.19114

[56] Lechen Zhang, Tolga Ergen, Lajanugen Logeswaran, Moontae Lee, and David Jurgens. 2024. SPRIG: Improving Large Language Model Performance by System Prompt Optimization. https://arxiv.org/abs/2410.14826

[57] Peng Zhang, Yang Wang, Xutong Liu, Yibiao Yang, Yanhui Li, Lin Chen, Ziyuan Wang, Chang ai Sun, and Yuming Zhou. 2022. Test suite effectiveness metric evaluation: what do we know and what should we do? https://arxiv.org/abs/2204.09165