2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) | 979-8-3315-3467-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSTW64639.2025.10962507

LLM Prompt Engineering for Automated White-Box Integration Test Generation in REST APIs

André Mesquita Rincon[†‡]
[†]*Campus Paraíso do Tocantins*
Federal Institute of Tocantins (IFTO)
Paraíso do Tocantins, Brazil
andrerincon@ifto.edu.br

Auri Marcelo Rizzo Vincenzi[‡]
[‡]*Department of Computing (DC)*
Federal University of São Carlos (UFSCar)
São Carlos, Brazil
auri@ufscar.br, andre.rincon@estudante.ufscar.br

João Pascoal Faria
INESC TEC, Faculty of Engineering,
University of Porto
Porto, Portugal
jpf@fe.up.pt

***Abstract*—This study explores prompt engineering for automated white-box integration testing of RESTful APIs using Large Language Models (LLMs). Four versions of prompts were designed and tested across three OpenAI models (GPT-3.5 Turbo, GPT-4 Turbo, and GPT-4o) to assess their impact on code coverage, token consumption, execution time, and financial cost. The results indicate that different prompt versions, especially with more advanced models, achieved up to 90% coverage, although at higher costs. Additionally, combining test sets from different models increased coverage, reaching 96% in some cases. We also compared the results with EvoMaster, a specialized tool for generating tests for REST APIs, where LLM-generated tests achieved comparable or higher coverage in the benchmark projects. Despite higher execution costs, LLMs demonstrated superior adaptability and flexibility in test generation.**

***Index Terms*—Prompt Engineering, Large Language Models, Automated Integration Testing, White-Box Testing, REST APIs.**

I. Introduction

Using web APIs (Application Programming Interfaces) in a globally distributed environment entails significant responsibility in developing and maintaining these interfaces, as a single failure can compromise thousands of services. In this context, two concepts stand out: REST APIs, which have become a popular approach to software development [1], and integration testing. A REST API is an interface that adheres to the principles of REST (Representational State Transfer) and enables communication between systems through a simple, scalable, and interoperable architecture [2]. Integration testing, conversely, ensures that interactions between different parts of the software function correctly [3].

REST APIs present unique challenges and opportunities for automated testing, driving the recent development of various techniques and tools for test case generation [4]. Manual test creation is labor-intensive and prone to mistakes, especially in distributed systems with multiple connections and information exchanges. In this scenario, recent advances in artificial intelligence have provided tools to address these challenges, particularly Large Language Models (LLMs) [5, 6]. These AI models are trained on vast amounts of text data to understand and generate natural language with human-like proficiency [7].

In this context, this work explores how to construct prompts for using LLMs in generating white-box integration test sets for REST APIs. We developed an automated script to generate test sets supported by OpenAI's LLM models [8]. We ran the script using four prompt versions and applied it to three different AI models across three projects employing REST APIs. Additionally, we compared the test sets obtained using the script and those generated by the EvoMaster[1] tool [9] to evaluate the performance and efficiency of LLMs relative to a specialized automated solution. This process addresses the following research questions:

- **RQ1:** How do different prompt versions influence the generation of white-box integration test sets for REST APIs using LLMs?
- **RQ2:** What improvements can be achieved by combining white-box integration test sets generated by different OpenAI models regarding code coverage (statement coverage) in REST API implementations?
- **RQ3:** What is the cost-benefit ratio of each LLM model in the automated generation of white-box integration tests for REST APIs?
- **RQ4:** What are the advantages and disadvantages of using LLMs for white-box integration test set generation for REST APIs compared to specialized tools?

The remainder of this work is as follows: Section II presents the theoretical foundation and reviews relevant concepts; Section III discusses related works; Section IV describes the development and execution of the experiment; Section V presents the collected data and analyses conducted to answer the research questions; Section VI addresses threats to validity, discussing potential limitations and variables that could affect the generalizability of the results; finally, Section VII concludes the study and outlines future research directions.

This research is partially supported by Brazilian Funding Agencies FAPESP (Grant nº 2019/23160-0 and 2023/00001-9), CAPES (Grant nº 001 and 88887.888653/2023-00), and CNPq.

[1]http://www.evomaster.org/

II. Background

This section briefly overviews three key concepts for understanding this study: software testing, REST APIs, and LLMs.

A. Software Testing

The testing activity dynamically verifies whether a program exhibits the expected behaviors for a finite set of test cases. Test case selection becomes critical, guided by risk and prioritization criteria, aiming to identify as many faults as possible [10]. It can be divided into three phases with distinct objectives: unit, integration, and system testing [3]. This paper targets integration testing. This type of testing requires a thorough understanding of the internal structures and interactions among components and, in general, is conducted by the development team, who possess deep knowledge of the system's internal dependencies [3].

Various techniques and criteria can be applied during testing, using either functional (black-box) approaches, which examine the external behavior of the software, or structural (white-box) approaches, which focus on the internal code structure [3]. Testing criteria establish testing requirements, which help testers select test cases to cover specific aspects of the software [3]. According to Wang et al. [11], code coverage is commonly adopted as a quality metric for test sets generated with the support of LLMs.

B. REST APIs

A REST API provides an interface based on the architectural principles of REST (Representational State Transfer). It facilitates communication between systems through a simple, interoperable, and scalable architecture [2], making it a common approach for exposing functionalities in contemporary systems [1]. REST operates on several fundamental principles [2]: (1) REST models all entities as **resources**, which clients manipulate using a limited set of operations; (2) Each resource uses a **unique identifier** through URLs (Uniform Resource Locators), enabling clients to access, create, update, or delete resources via their URLs; and (3) REST maintains a **stateless** structure, where each client request includes all the information required for the server to process it.

In a REST API implementation, clients transfer resource representations identified by URLs and manipulate them using HTTP (Hypertext Transfer Protocol) methods, often called HTTP verbs, such as GET, POST, PUT, and DELETE [12]. HTTP status codes play a crucial role in REST APIs by informing clients about the outcomes of their requests and guiding subsequent actions. These codes convey results across five categories: 1xx for informational responses, 2xx for successes, 3xx for redirection, 4xx for client errors, and 5xx for server errors [13].

Developers frequently use the OpenAPI Specification (OAS)[2] to document and specify resources, available operations, and expected status codes in REST APIs.

[2]https://swagger.io/specification/

C. Large Language Models (LLMs)

Large Language Models (LLMs) use extensive datasets to capture patterns and structures in language, enabling them to learn from complex and diverse training data. Their ability to generate human-like text and their applications in various domains, including software engineering, have increased significantly, especially on testing [11].

Most LLMs rely on the Transformer architecture, which includes an encoder for input representation and a decoder for output generation. OpenAI's Generative Pre-trained Transformer (GPT) stands out for its instruction tuning and reinforcement learning with human feedback (RLHF). These features enable GPT to address code generation and comprehension tasks effectively [6].

Prompt engineering plays a crucial role in maximizing LLM performance. Researchers have identified five primary techniques: zero-shot learning, few-shot learning, chain-of-thought (CoT) prompting, self-consistency, and automated prompting [11]. In zero-shot learning, users provide the model with only a task description, a technique often applied to generate unit tests. Few-shot learning enhances performance for complex tasks by including high-quality examples. CoT and self-consistency focus on sequential reasoning and hypothesis validation, respectively, while automated prompts iteratively optimize instructions [11].

Frameworks and guidelines such as TELeR (Turn, Expression, Level of Details, Role) and OpenAI's documentation provide structured approaches for designing prompts. TELeR categorizes and designs prompts for complex tasks based on four dimensions: the number of interactions required (Turn), the expression style of instructions (Expression), the level of detail (Level of Details), and role definition within the system (Role). This framework proves particularly relevant for GPT's performance in tasks requiring advanced semantic understanding and involving multiple subprocesses [14].

According to OpenAI [8], prompt creation combines system and user prompts to guide the model effectively. The system prompt establishes the general context and role of the model, defining its responsibilities and expected response style. The user prompt specifies the task, including clear instructions, success criteria, and examples or input data when necessary. By combining both, the model receives detailed guidelines, maximizing the accuracy and relevance of responses.

III. Related Work

Researchers have extensively explored using LLMs in software testing, as demonstrated in the review by Wang et al. [11]. This study analyzed 102 works on applying LLMs across various stages of the testing lifecycle, focusing on tasks such as generating unit test cases, test oracles, system test inputs, debugging, and program repair. The authors highlighted vital challenges and opportunities in this field, including improving test coverage, addressing the test oracle problem, and applying LLMs to real-world scenarios. The study identified the potential of advanced prompt design techniques and integration testing. In addition to this broad review, researchers have explicitly concentrated on using LLMs to test REST APIs [15, 16, 17, 18, 19].

Corradini et al. [15] introduced DeepREST, a tool that uses machine learning to automate black-box testing for REST APIs. DeepREST overcomes the limitations of traditional tools that rely solely on OAS for extracting syntactic information about APIs. The authors leveraged LLMs to generate realistic input parameter values based on parameter names, OAS descriptions, and endpoint contexts.

Huynh et al. [16] proposed a method to address the limitations of RESTful API testing that depends exclusively on OAS. Unlike traditional approaches focusing on status code and schema validation, APITesting uses LLMs to interpret natural language descriptions in API specifications. It enables mining and validating complex business logic and data integrity constraints. This method achieves 94.3% accuracy in constraint identification and 88.5% in automated test generation.

Kim et al. [17] proposed RESTGPT, a method that uses the contextual capabilities of LLMs to enhance REST API testing. RESTGPT overcomes the limitations of traditional keyword-based techniques and manual validations by extracting machine-readable rules from natural language API specifications and generating contextualized parameter values. This approach eliminates the need for exhaustive validations while delivering up to 97% accuracy.

Pereira et al. [18] developed APITestGenie, a tool that automates the generation of executable test scripts from business requirements and API specifications. APITestGenie generates tests that simulate real-world scenarios, often involving multiple endpoints, and enables integration tests to validate interactions and data flows between services. In experiments with ten real APIs, APITestGenie achieved an initial 57.3% success rate for generating valid scripts, which increased to 80% after three attempts per task.

Sri et al. [19] presented a method that generates automated test cases by creating executable scripts from business requirements and API specifications in the Postman[3] format using LLM. The authors extracted parameter dependencies and schema validations from natural language API descriptions and generated detailed scenarios to improve test coverage.

Despite these advancements, this paper provides distinct contributions. It demonstrates the impact of prompt engineering strategies on the quality of generated tests and adopts a white-box testing approach by integrating the application's source code and OAS. The paper also compares the performance of different GPT models regarding code coverage, token consumption, and execution time, benchmarking them against EvoMaster. Finally, the work analyzes the costs of using LLMs, highlighting the cost-benefit ratio.

IV. Experiment Design

The experiment investigated different prompt versions and LLM models for automatically generating white-box integration test sets for REST APIs. We used EvoMaster test sets as a baseline for comparison.

[3]https://www.postman.com

The decision to include EvoMaster in this experiment stems from its status as one of the most advanced tools supporting white-box testing for RESTful APIs. EvoMaster combines search-based software testing (SBST) approaches with fuzzing techniques [20]. It uses testing heuristics to explore the internal structure of applications and generate test scenarios. EvoMaster stands out as a pioneer in the RESTful API domain, underscoring its relevance and comprehensiveness as a suitable choice for achieving the objectives of this study [4].

The experiment followed two main stages: (1) running EvoMaster to generate native test sets and (2) developing a Python[4] script that integrates the OpenAI API[5] to automate test generation with support of LLM models. The script also uses Maven[6] for compiling and executing the generated tests, ensuring a seamless and automated workflow.

Integrating the Python script with the OpenAI API enabled GPT-3.5 Turbo, GPT-4 Turbo, and GPT-4o models. We selected these models due to their accessibility via API, which allowed automation and replication of processes for generating and refining the test sets. This API access also provided the means to monitor metrics such as token consumption and execution time. We use the JaCoCo[7] tool to measure statement coverage of the analyzed projects.

The experiments used three projects from the EvoMaster Benchmark (EMB)[8] – Market, Ncs, and Scs. The market is a real project for managing e-commerce with 13 endpoints and 327 LOCs[9]. The Ncs and Scs are artificial projects designed for REST API experiments. The Ncs implements numerical algorithms (6 endpoints and 605 LOCs), and The Scs implements string manipulation (11 endpoints and 862 LOCs). We chose these projects because they made analysis feasible within the experiment's financial constraints. We configured EvoMaster to generate test sets for these projects, creating a reference baseline for comparing code coverage between EvoMaster and the LLMs-generated test sets.

The experiment included four versions of prompts (v0, v1, v2, and v3). Each version incorporated elements and instructions to investigate how language models respond to prompt variations when generating white-box integration tests for REST APIs. The prompts use a few-shot learning approach, where examples guide the models' behavior. Each prompt version includes task descriptions to help the models generate test sets with greater precision and alignment with the experiment's objectives.

The script generated three test sets per model and project for each prompt version, producing 27 test sets for all projects for one single prompt version. We ran the script on alternate days to minimize threats, according to guidelines from [21]. The data collection recorded code coverage, token consumption, and execution time.

[4]https://www.python.org
[5]https://platform.openai.com/docs/api-reference
[6]https://maven.apache.org
[7]https://www.eclemma.org/jacoco/
[8]https://github.com/WebFuzzing/EMB
[9]https://javancss.github.io/

We configured EvoMaster to execute ten consecutive runs for each project with different seeds to improve variability. The Python script executed three runs per model and project due to budget constraints, as the Cost of using the OpenAI API increases with the volume of generated data. We set the OpenAI models' temperature parameter to 0.7, following OpenAI's recommendations for balancing consistency and creativity [8].

V. Data Collection and Analysis

This section presents the collected data and the analyses conducted to answer the research questions.

A. Research Question 1 (RQ1)

To address this question, we developed four distinct prompt versions based on practices from the literature. The prompts' content and the results obtained for each generated test set are detailed below.

We designed the first prompt using an ad-hoc approach, following an iterative trial-and-error process [22]. The primary challenge involved creating prompts that provided sufficient information for GPT models to generate suitable tests that met the specified requirements while incorporating the necessary EvoMaster fixture code for execution with Maven. After refining, we reached prompt version v0 (Figure 1).

```
1   System prompt:You are an experienced test engineer specializing
2   in code coverage, REST API testing, and integration testing.
```

```
1   User prompt: Create a white box test set in JUnit 5 and
2   Java 11 using REST Assured for the Java code delimited by *****.
3   Use the swagger documentation delimited by ##### to provide
4   complementary information about the application. The test set
5   should be created to maximize coverage (decision coverage) and
6   cover the specification and all possible responses (status code).
7   The response must contain only Java code in JUnit 5 format.
8
9   If necessary, use these data to registered user:
10  {"email":"ivan.petrov@yandex.ru","password":"petrov",
11  "name":"Ivan Petrov","phone":"+7 123 456 78 90",
12  "address":"Riesstrasse 18"}
13
14  Follow these five instructions for this generated test code:
15  //EvoMaster code instructions (1 to 5)
16
17  *****
18  {java_code}
19  *****
20
21  #####
22  {openapi_specification}
23  #####
```

Fig. 1. Prompt version v0 for API test set generation

We simplify the system prompt to include only the persona. The user prompt included a base directive, authentication data, instructions, EvoMaster fixture code to configure the application, application source code (`{java_code}`), and the OpenAPI specification (`{openapi_specification}`). We omitted the last three items in Figures 1, 2, 3, and 4 due to space constraints. The complete versions of the prompt can be found elsewhere[10].

In creating prompt version v1 (Figure 2), we followed the TELeR taxonomy guidelines [14] and OpenAI's prompt design recommendations [8].

Based on these principles, we expanded the system prompt in version v1 to include the following: a more detailed description of the LLM's role, specifics on how the tests should be generated, criteria for evaluating the success of the generated tests, and an emphasis on the logical correctness of the tests and the quality of the generated code. The user prompt in version v1 retained the elements from v0, except that it excluded instructions related to code coverage.

```
1   System prompt: You are an experienced test engineer specialized
2   in creating white box test sets in JUnit 5 using REST Assured
3   and Java 11.
4
5   Your task is to create test sets that maximize decision
6   coverage, adhere to API specifications, and validate all
7   possible responses, including status codes. The response must
8   contain only Java code in JUnit 5 format without explanations.
9
10  The response should follow this structure:
11  1. Package declaration.
12  2. Import statements.
13  3. Class declaration.
14  4. Variable declarations.
15  5. Setup and teardown methods.
16  6. Test methods organized according to the API endpoints being
17  tested.
18
19  The test generation will be assessed on several key factors:
20  1. Executability: The tests must run smoothly and without errors.
21  2. Relevance: The tests must be meaningful and appropriate for
22  the code and swagger documentation.
23  3. Correctness: The test code should be free of logical errors.
24  4. Coverage: The tests should cover as many endpoints and
25  scenarios as possible to ensure thorough validation.
26  5. Code Quality: The code is reliable, executable, and includes
27  clear explanations where necessary.
28  6. Endpoint Accuracy: The API call matches the type of the
29  request and response object.
```

```
1   User prompt: Create a white box test set in JUnit 5 and
2   Java 11 using REST Assured for the Java code delimited by *****.
3   Use the swagger documentation delimited by ##### to provide
4   complementary information about the application.
5
6   // Same as v0 User Prompt - Lines 9 to 23
```

Fig. 2. Prompt version v1 for API test set generation

The development of prompt version v2, compared to v1, included additional instructions related to testing error scenarios (status code 5xx), validating OpenAPI Schema[11], and addressing business rules in both the user and system prompts.

For the user prompt, version v2 retained all elements from v0, but it added instructions for detecting errors (status code 5xx), validating the OpenAPI Schema, and addressing business rules. Figure 3 presents the final version of prompt v2.

We developed prompt version v3 by considering the previous versions and their results. The goal for v3 focused on combining the "creativity" of the user prompt from v0 with the stability of a system prompt that guides outcomes without imposing excessive rules, as seen in v2. We retained the exact structure from version v2 for the system prompt, as it did not restrict the model's creativity with rigid rules and achieved good code coverage results.

For the user prompt, we used the structure of v0 as the base, given that v0 delivered high average coverage and the highest individual coverage in the two projects. The primary difference between v3 and v0 was adding a request to generate test data on error detection (status code 5xx). Figure 4 shows the final version of prompt v3.

We applied each prompt version to three GPT models for the three analyzed projects. Consequently, each prompt version generated three test sets per model/project tuple, resulting in nine test sets per model and nine per project. This setup

[10]https://github.com/aurimrv/api_testing/

[11]https://swagger.io/docs/specification

produced 27 test sets per prompt version, totaling 108 test sets for the experiment. We collected the following data for each test set: code coverage, token consumption during test generation, and the time required to generate each set.

```
System prompt: You are an experienced test engineer specializing
in REST API testing, integration testing, and code coverage analysis
using JUnit 5, Java 11, and REST Assured. Your task is to create a
comprehensive test set that maximizes code coverage, including
decision coverage and thorough validation of API responses. The
test set must consist of tests that validate the robustness of
the API against internal errors, compliance with the OpenAPI
schema, and adherence to business rules.

1. Error Detection: Include tests that force the API to return
status codes in the 5xx range (e.g., 500 Internal Server Error)
by simulating invalid inputs or server failures. The tests should
include assertions to capture these status codes as indicators of
failures.

2. Schema Validation: Ensure that all API responses conform to the
OpenAPI schema. This includes checking for required fields, data
type conformity, and validity of returned values. Any discrepancy
between the response and the schema should be flagged as a failure.

3. Business Rule Enforcement: Validate that all business rules
defined by the API are being followed. This includes checking
that POST, PUT, and DELETE operations behave as
expected|creating, modifying, or deleting resources correctly.
Any violation of these rules should be detected and reported.

While adhering to the structure provided, feel free to explore
various approaches to testing, including edge cases, error
handling, and schema validation. The goal is to cover as many
different scenarios as possible to ensure a robust and
comprehensive test set.

The generated test set must contain only Java code in JUnit 5
format, with no additional explanations.
```

```
User prompt: Create a white box test set in JUnit 5 and
Java 11 using RESTAssured for the Java code delimited by *****.
Use the swagger documentation delimited by ##### to provide
complementary information about the application. The objective
of this test set is to maximize code coverage (including
decision coverage), ensure compliance with the API specification,
and validate the robustness, schema conformity, and business rule
enforcement of the API.

// Same as v0 User Prompt - Lines 9 to 15

Instruction 6. Test Methods: Ensure the generated tests include:
* Error Scenarios: Tests that simulate invalid inputs or
conditions that force the API to return 5xx status codes,
particularly 500. Include assertions to verify these responses.
* Schema Validation: Tests that check whether the API responses
conform to the OpenAPI schema. Ensure all required fields are
present, data types match, and values are valid.
* Business Rule Enforcement: Tests that validate operations
like POST, PUT, and DELETE follow the business rules, ensuring
correct creation, modification, or deletion of resources. Any
violations should be captured and asserted.

\vspace{-3mm}
// Same as v0 User Prompt - Lines 17 to 23
```

Fig. 3. Prompt version v2 for API test set generation

```
System prompt: // Same of system prompt v2 (Figura 4)
```

```
User prompt: Create a white box test set in JUnit 5 and
Java 11 using REST Assured for the Java code delimited by *****.
Use the swagger documentation delimited by ##### to provide
complementary information about the application. The test set
should be created to maximize coverage (decision coverage) and
cover the specification and all possible responses (status code).
Ensure the generated tests include tests that simulate invalid
inputs or conditions that force the API to return 5xx status
codes (include assertions to verify these responses).
\vspace{-3mm}
// Same as v0 User Prompt - Lines 9 to 23
```

Fig. 4. Prompt version v3 for API test set generation

Table I organizes the collected data as follows: the column "GPT Models" lists the GPT models used; the column "Prt" details the prompt versions designed for this study (v0 to v3); and for each analyzed project (Market, Ncs, and Scs), the table presents the maximum code coverage in percentage (MCV) achieved by merging the test sets generated from three separate executions, the sum of token consumption in thousands (STK), and sum of time spent in seconds (STM) across the three test sets for each model/prompt version/project combination. We calculated token consumption and execution time by summing the recorded values from the corresponding test set generations. Full data for each prompt version is in its complete form in the repository containing the experimental data[12]. The table highlights the best prompt results, considering coverage, token consumption, and execution time, independently of the GPT model.

TABLE I
MAXIMUM CODE COVERAGE, SUM OF TOKEN CONSUMPTION, AND SUM OF EXECUTION TIME FOR EACH MODEL/PROMPT VERSION/PROJECT.

GPT Models	Prt	Market			Ncs			Scs		
		MCV	STK	STM	MCV	STK	STM	MCV	STK	STM
3.5 Turbo	v0	51.2	273.1	1040	55.2	353.2	1146	79.2	454.8	1395
	v1	56.2	414.8	1479	67.4	325.4	1150	73.0	436.4	1306
	v2	56.0	444.0	1373	78.9	258.0	898	33.7	444.9	1507
	v3	59.7	308.5	1082	64.1	254.2	862	68.1	762.9	2118
4 Turbo	v0	90.4	358.0	2437	92.0	226.4	1710	92.1	435.1	3145
	v1	84.6	317.6	1752	58.3	201.8	2195	84.4	448.2	2339
	v2	83.8	378.6	2491	67.2	281.7	1820	83.5	555.3	2776
	v3	90.4	356.5	2281	89.1	245.0	1648	84.0	511.3	2723
4o	v0	74.2	713.9	2377	84.5	257.7	896	88.6	577.1	1627
	v1	80.1	421.4	1666	71.5	274.8	1128	83.5	645.7	1951
	v2	86.0	611.1	1872	92.0	340.5	1049	90.6	889.4	2699
	v3	85.9	733.9	2307	90.3	308.7	1001	89.0	774.6	2661

In an ideal scenario, we want a prompt that reaches the highest coverage and consumes fewer tokens and time. However, as can be observed in Table I, the results from each test set generated by different prompt versions vary depending on the GPT model and the project, meaning there is no single best prompt version for all cases.

Version v0 achieved the highest code coverage across all projects using GPT-4 Turbo, reaching 90.4% in Market, 92.0% in Ncs, and 92.1% in Scs, while also showing the lowest token consumption and shortest execution time with GPT-3.5 Turbo in the Market project. Version v1 did not excel in coverage but stood out for low token consumption with GPT-4 Turbo in Ncs (201.8K) and short execution time with GPT-3.5 Turbo in Scs (1306 seconds). Versions v2 and v3 also performed well, with v2 achieving 92.1% in Ncs (GPT-4o) and v3 reaching 90.4% in Market (GPT-4 Turbo) and the shortest time, 862 seconds, in Ncs (GPT-3.5 Turbo).

The coverage results achieved with versions v0, v2, and v3, especially with the GPT-4 Turbo and 4o models, and the poor performance of version v1 suggest that providing a very rigid set of instructions in the system prompt, as in v1, can negatively impact test case generation. Token consumption and execution time varied across the results, so they do not directly correlate with prompt structure, meaning that these factors may depend on other elements that are not directly related to prompt structure. It is important to note that prompts v2 and v3 include instructions to verify API responses about the 5xx status code. However, this did not result in a higher coverage.

Answer to RQ1: Different prompt versions significantly influence the generated test sets. Versions with less rigid instructions in the system prompt achieve better code coverage results, especially with the latest models.

[12]https://github.com/aurimrv/api_testing/

B. Research Question 2 (RQ2)

To address this question, we analyzed the results of the different prompt versions by combining the test sets generated by each model for each project. We performed the model combinations by merging the test sets from each model/prompt version/project combination. The analyzed combinations included the following configurations: 3.5+4, 3.5+4o, 4+4o, and 3.5+4+4o. This analysis evaluated whether combining tests generated by different models could increase code coverage by leveraging the test sets' potential complementarity.

Tables II presents the combined data for each prompt version. These tables follow the structure: the "GPT Models" column lists the individual models and their combinations, and for each project (Market, Ncs, and Scs), the tables display maximum code coverage (MCV) achieved by merging the test sets generated from three separate executions. For the Market and Scs project, combinations provide no coverage improvement for v0. In contrast, the Ncs project showed improved code coverage on combinations (highlighted cells): 3.5+4, an improvement of 0.9%; 4+4o, an improvement of 1.6%, and 3.4+4+4o with an improvement of 2.5%, face the best coverage from individual model 4 of 92.0%.

TABLE II
COMBINED RESULTS FOR PROMPT VERSIONS.

GPT Models	Market (MCV)				Ncs (MCV)				Scs (MCV)			
	v0	v1	v2	v3	v0	v1	v2	v3	v0	v1	v2	v3
3.5	51.2	56.2	56.0	59.7	55.2	78.9	64.1	67.4	79.2	73.0	33.7	68.1
4	90.4	84.6	83.8	90.4	92.0	58.3	67.2	89.1	92.1	84.4	83.5	84.0
4o	74.2	80.1	86.0	85.9	84.5	71.5	92.0	90.3	88.6	83.5	90.6	89.0
3.5+4	90.4	84.6	88.2	90.4	92.9	70.1	84.4	94.3	92.1	85.7	83.5	84.0
3.5+4o	74.2	80.1	90.4	85.9	85.4	73.7	95.3	95.1	88.8	83.7	90.6	89.2
4+4o	90.4	90.4	86.0	91.7	93.6	73.6	93.4	90.8	92.1	87.3	90.7	89.9
3.5+4+4o	90.4	90.4	90.4	91.7	94.5	74.1	96.3	95.1	92.1	87.3	90.7	89.9

The v1 prompt for the Market project, combinations with GPT-4 Turbo increased code coverage to 90.4%. For the Ncs project, most model combinations improved coverage, ranging from 73.6% to 74.1%. In the Scs project, coverage also improved across all combinations, varying from 83.7% to 87.3%. The v2 prompt reveals that most model combinations improved coverage, ranging from 88.2% to 90.4% for the Market project and from 93.4% to 96.3% for project Ncs, once again demonstrating that the combinations enhanced coverage. In the Scs project, we got a minor increment of 0.1% when combining GPT-4 Turbo and GPT-4o.

Finally, the v3 prompt reveals that the combination of GPT-4 Turbo and GPT-4o was responsible for the improvements in the Market project, which reached 91.7%. In the Ncs project, all model combinations improved coverage, ranging from 90.8% to 95.1%, and in the Scs project, combinations improved coverage from 89.2% to 89.9%.

Answer to RQ2: Combining models has the potential to complement code coverage. However, in many cases, the models explored the same paths, resulting in limited variation in total coverage. Future optimizations could involve using models incrementally, leveraging more sophisticated models to cover areas missed by simpler ones, reducing overlap, and improving cost-effectiveness.

C. Research Question 3 (RQ3)

To answer this question, we analyzed financial and operational aspects, including "Cost" in dollars, token consumption, and the number of tests generated by each model.

Table III provides a detailed view of the collected data, presenting the following columns: "GPT Model"; "Dashboard Cost", indicating the total cost in dollars recorded on the OpenAI dashboard; "Tokens Consumed", showing the total tokens used by each model in thousands, as detailed in Table I; "Cost per Tokens", an estimate of the cost per token in dollars; "Test Set", representing the number of test cases generated by each model considering all executions; "Cost per Test", calculating the estimated cost of generating each test case; and "Average Tokens per Test", showing the average tokens required to create a single test case.

TABLE III
FINANCIAL COST OF THE EXPERIMENT

GPT Model	Dashboard Cost	Tokens Consumed	Cost per Token	Test Set	Cost per Test	Average Tokens per Test
3.5	$3.62	4730.4	$0.000000765	950	$0.00381053	4979.3
4	$49.08	4315.6	$0.000011373	1304	$0.03763804	3309.5
4o	$53.15	6743.3	$0.000007882	2476	$0.02146607	2723.5

The data shows that GPT-3.5 Turbo emerged as the most economical option, with a total cost of only $3.62 and a low cost per test ($0.00381053). In contrast, the more advanced models, GPT-4 Turbo and GPT-4o, incurred significantly higher total costs of $49.08 and $53.15, respectively.

Both advanced models generated more tests than GPT-3.5 Turbo, with 1,304 tests for GPT-4 Turbo and 2,476 tests for GPT-4o. Additionally, GPT-4o demonstrated efficiency in token consumption per test, averaging 2,723.47 tokens per test, the lowest among the analyzed models. This efficiency suggests that GPT-4o processes prompt effectively and generates concise, focused responses, optimizing token usage.

However, the cost per token for GPT-4 Turbo ($0.0000011373) was significantly higher than that of GPT-3.5 Turbo ($0.000000765) and GPT-4o ($0.0000007882). This discrepancy contributed to GPT-4 Turbo's higher total costs. Although this model provided good code coverage (see Subsection V-A), cost per test ($0.03763804) makes it less appealing for budget-constrained projects.

Answer to RQ3: GPT-3.5 Turbo provided the best financial cost-benefit ratio, while GPT-4 Turbo and GPT-4o delivered higher test coverage and generation efficiency at significantly higher costs. Combining cheaper models for common code areas and reserving advanced models for uncovered parts could reduce costs and minimize overlaps to optimize usage.

D. Research Question 4 (RQ4)

To answer this question, we analyzed the results of the test sets generated by EvoMaster individually and combined them with the tests generated by GPT models.

We also ran EvoMaster to generate tests for the three projects: Market, Ncs, and Scs. The experiment included ten executions of one hour each for every project, following a practice based on scientific studies, where one-hour runs provided significant results [20]. After that, we measure maximum code coverage by merging the ten test sets. Besides that, we combine all tests from EvoMaster and GPT Models.

Table IV shows the maximum coverage reached for projects by EvoMaster in ten runs and combined with the tests generated by GPT models. Furthermore, the maximum combined GPT model coverage (from Tables II) is shown to facilitate comparison.

TABLE IV
RESULTS OF THE TEST SETS GENERATED BY EVOMASTER INDIVIDUALLY AND COMBINED WITH GPT MODELS

Maximum Combinations	**Market**	**Ncs**	**Scs**
EvoMaster (ten runs)	85.5	96.8	91.8
GPT 3.5+4+4o	91.7	96.3	92.1
EvoMaster+GPT	91.7	97.1	92.1

The first aspect in Table IV is that GPT combined test sets overcome EvoMaster in two out of three projects: in the case of the Market project, the improvement was 6.2%, and in the case of the Scs project, coverage improved 0.3%. In the case of the Ncs project, EvoMaster overcomes GPT combined 0.5%.

In the last line of Table IV, we presented the best results by combining all generated test sets (EvoMaster+GPT). In this case, the Market and Scs projects had no improvement, but in the case of the Ncs project, the coverage reached the highest value of 97.1%.

These coverage results indicate that LLMs provide a viable alternative for test generation, especially when considering the costs associated with developing and maintaining a tool like EvoMaster, which supports specific languages and platform versions [9]. Consequently, updates or changes in the platforms used require continuous maintenance.

In contrast, LLMs operate independently of language and platform, allowing users to apply them to any environment as long as they configure the prompts correctly. Additionally, using LLMs avoids the complexity of configuring and integrating a tool like EvoMaster into each specific project, which often requires significant time and specialized technical knowledge.

Answer to RQ4: LLMs provide a viable alternative for automated test generation with faster execution times and platform independence. However, their cost and token consumption may limit scalability compared to dedicated tools like EvoMaster. Combining different models and prompts may minimize the drawbacks of improving test set quality using an incremental testing strategy.

VI. Threats to Validity

Sallou et al. [21] propose guidelines to enhance validity in LLM-based software engineering research. Key recommendations include performing multiple inferences to track variability, documenting detailed metadata, using semantic-preserving metamorphic transformations, increasing project diversity, and comparing open and closed-source LLMs.

One potential factor that could compromise the validity of this study is the execution of three iterations per model/prompt version/project combination. It reduces the capacity for statistical analysis and may not fully capture the models' behavior under varying conditions, though it was necessary due to budget constraints.

Another significant threat is the limited number of REST API projects used in the experiment. Although selecting smaller projects enabled execution within financial constraints, it may restrict the generalization of the results to more complex and diverse applications.

Furthermore, the structure of the prompts and the variations created during the study threaten construct validity, as incremental changes in the prompts can influence the results concerning code coverage and execution time.

Finally, the exclusive use of closed-source models from OpenAI presents a critical limitation, as it prevents broad comparisons with open-source models.

VII. Conclusions and Future Work

This study investigated the design of prompts for generating white-box integration test sets for REST APIs using LLMs. The research focused on constructing different prompt versions to evaluate how incremental changes impact code coverage, execution time, and finances based on token consumption.

The study advances state-of-the-art automated integration testing with LLMs, providing theoretical and practical foundations for prompt engineering while proposing strategies that balance cost and performance. These findings reinforce the potential of LLMs as a promising approach for this purpose.

The research included comparative analyses of different OpenAI models, highlighting the importance of strategies that combine models incrementally to optimize cost-effectiveness. Additionally, the comparison with the EvoMaster tool underscored the potential of LLMs to offer complementary and, in some cases, superior solutions compared to specialized tools.

Key contributions of this study include the practical application of prompt engineering for test generation, the investigation of complementarity among different GPT models, and the analysis of how to integrate diverse sources, such as OAS specifications and source code, to enhance API validation. The results demonstrate that combining models can improve code coverage, though this approach incurs additional costs.

Future work will explore incorporating open-source LLMs to compare their performance and cost with closed-source models, including more diverse projects across domains and sizes, and analyzing the quality of generated test sets based on criteria such as defect detection, errors, failures, and test quantity.

Acknowledgment

This work is partially supported by Brazilian Funding Agencies FAPESP (Grant nº 2019/23160-0 and 2023/00001-9), CAPES (Grant nº 001), and CNPq.

References

[1] S. Karlsson, R. Jongeling, A. Čaušević, and D. Sundmark, "Exploring behaviours of restful apis in an industrial setting," *Software Quality Journal*, vol. 32, no. 3, p. 1287–1324, Jul. 2024. [Online]. Available: https://doi.org/10.1007/s11219-024-09686-0

[2] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[3] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 2008.

[4] A. Golmohammadi, M. Zhang, and A. Arcuri, "Testing restful apis: A survey," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, Nov. 2023. [Online]. Available: https://doi.org/10.1145/3617175

[5] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.

[6] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660783

[7] A. Al-Kaswan and M. Izadi, " The (ab)use of Open Source Code to Train Large Language Models ," in *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 9–10. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/NLBSE59153.2023.00008

[8] OpenAI, "Openai api reference," Available at: https://platform.openai.com/docs/api-reference, 2024, accessed on: Mar. 08, 2024.

[9] A. Arcuri, "Evomaster: A tool for automatically generating system-level test cases," Available at: https://github.com/WebFuzzing/EvoMaster, 2024, accessed on: Nov. 15, 2023.

[10] H. e. Washizaki, *Guide to the Software Engineering Body of Knowledge*, 4th ed. IEEE Computer Society, 2024, accessed on: Oct. 18, 2024.

[11] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.

[12] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, 2013.

[13] L. Gupta, "Rest api tutorial," Available at: https://restfulapi.net, 2024, accessed on: Sep. 09, 2024.

[14] S. K. K. Santu and D. Feng, "Teler: A general taxonomy of llm prompts for benchmarking complex tasks," 2023. [Online]. Available: https://arxiv.org/abs/2305.11430

[15] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, "Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1383–1394. [Online]. Available: https://doi.org/10.1145/3691620.3695511

[16] H. Huynh, Q.-T. Le, T. N. Nguyen, and V. Nguyen, "Using llm for mining and testing constraints in api testing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2486–2487. [Online]. Available: https://doi.org/10.1145/3691620.3695341

[17] M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, "Leveraging large language models to improve rest api testing," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 37–41. [Online]. Available: https://doi.org/10.1145/3639476.3639769

[18] A. Pereira, B. Lima, and J. P. Faria, "Apitestgenie: Automated api test generation through generative ai," 2024. [Online]. Available: https://arxiv.org/abs/2409.03838

[19] S. D. Sri, M. A. S, S. V. R, R. C. Raman, G. Rajagopal, and S. T. Chan, "Automating rest api postman test cases using llm," 2024. [Online]. Available: https://arxiv.org/abs/2404.10678

[20] M. Zhang and A. Arcuri, "Open problems in fuzzing restful apis: A comparison of tools," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, Sep. 2023. [Online]. Available: https://doi.org/10.1145/3597205

[21] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 102–106. [Online]. Available: https://doi.org/10.1145/3639476.3639764

[22] N. Shao, Z. Cai, C. Liao, Y. Zheng, Z. Yang *et al.*, "Compositional task representations for large language models," in *The Eleventh International Conference on Learning Representations*, 2023.