Informatics

Automating Software and Vulnerability Testing in REST APIs with Large Language Models

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software und Internet Computing

eingereicht von

Diana Strauss, BSc
Matrikelnummer 01526223

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl. Ing. Dr. sc. Jürgen Cito, BSc.

Mitwirkung: DI Andreas Happe, BSc

Wien, 26. Februar 2025

Diana Strauss

Jürgen Cito

Informatics

Automating Software and Vulnerability Testing in REST APIs with Large Language Models

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software and Internet Computing

by

Diana Strauss, BSc
Registration Number 01526223

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl. Ing. Dr. sc. Jürgen Cito, BSc.
Assistance: DI Andreas Happe, BSc

Vienna, February 26, 2025

Diana Strauss

Jürgen Cito

Erklärung zur Verfassung der Arbeit

Diana Strauss, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 26. Februar 2025

Diana Strauss

Danksagung

Bevor ich mein Masterstudium abschließe, möchte ich mir einen Moment nehmen und all jenen danken, die mich auf diesem Weg unterstützt haben.

Als erstes möchte ich meiner Mutter danken, die mir beigebracht hat, dass alles möglich ist, wenn man hart arbeitet. Meinen besonderen Dank möchte ich meiner Oma, meinem Vater und meinem Freund aussprechen, da sie mich immer dazu ermutigt haben, niemals aufzugeben und stets nach vorn zu schauen – auch in schwierigen Zeiten. Ich habe meinen Großeltern einen großen Dank für ihre Hilfe zukommen lassen, indem sie mir die Wertschätzung von Bildung und die Bedeutung der Möglichkeit, sie zu erhalten, nähergebracht haben.

Ein großes Dankeschön geht an meinen Betreuer, Jürgen Cito, und meinen stellvertretenden Betreuer, Andreas Happe. Ohne ihre Unterstützung und Anleitung wäre diese Arbeit – und die dahinterstehende Forschung – nicht möglich gewesen. Außerdem bin ich sehr dankbar, dass sie mich für dieses faszinierende Themengebiet begeistert haben.

Acknowledgements

As I approach the end of my master's program, I want to take a moment to express my gratitude to everyone who has been a part of this journey.

First, I want to say thank you to my mother for teaching me if you belive in yourself and work hard that anything is possible. A special thank you to my grandmother, father and boyfriend for their constant encouragement to never give up and to always look ahead, even in the face of challenges. I also owe a debt of gratitude to my grandparents, who taught me the value of education and the importance of being thankful for the opportunity to receive it.

A big thank you to my supervisor, Jürgen Cito, and my assistant supervisor, Andreas Happe. Without their guidance and support this thesis and the research behind it would not have been possible. Moreover, I am incredibly grateful to them for introducing me to this fascinating field.

Kurzfassung

Diese Arbeit untersucht die Wirksamkeit von Large Language Models (LLMs) in der automatisierten Testfallgenerierung für REST-APIs. Der Fokus liegt auf ihrer Leistungsfähigkeit, den Auswirkungen von Prompt Engineering-Techniken und ihrer Fähigkeit, Schwachstellen zu identifizieren und die Testabdeckung zu verbessern. REST-APIs sind aufgrund ihrer Einfachheit essenziell für die Webentwicklung, jedoch anfällig für Angriffe, weshalb umfassende Tests unerlässlich sind. LLMs können helfen die API-Dokumentation und Testprozesse zu optimieren und somit die Sicherheit und Belastbarkeit erhöhen. Ziel dieser Forschung ist es einen LLM-basierten Ansatze zu entwickeln, der Testfälle generiert, die Abdeckung verbessert und gleichzeitig die Testkosten minimiert.

Die Arbeit stellt zwei neue Anwendungsfälle vor: WebAPIDocumentation und WebAPITesting. Die Arbeit zielt darauf ab, die Erstellung von OpenAPI-Spezifikationen zu automatisieren, Dokumentationsfehler zu beheben und Tests zur Identifikation von Schwachstellen wie fehlerhafter Authentifizierung durchzuführen. Die Ergebnisse zeigen, dass GPT-4o-mini o1 effektiver Endpunkten und Parametern erkennt. Das ist insbesondere der Fall, wenn die Prompts mit Prompt Engineering-Techniken verwendet werden wie Tree-of-Thought(TOT),Chain-of-Thought (COT) und In-Context Learning (ICL). Obwohl GPT-4o-mini sich beim Erkennen gängiger Schwachstellen wie fehlerhafter Authentifizierung und NoSQL-Injection auszeichnet, stößt es bei komplexeren Problemen wie der Offenlegung sensibler Daten und unzureichender Ratenbegrenzung auf Schwierigkeiten. Die Ergebnisse zeigen das Potenzial von LLMs zur Verbesserung von Sicherheitstests und betonen die Rolle des Prompt Engineering bei der Verbesserung der Effizienz und Genauigkeit automatisierter Tests. Insgesamt trägt diese Arbeit zum aktuellen Fachgebiet bei, indem sie zeigt, dass LLMs zum Generieren von Testfällen für REST-APIs verwendet werden können und dabei effizient sind. Ihre Leistung kann durch den Einsatz von Prompt Engineering-Strategien wie COT verbessert werden. Darüber hinaus zeigte sie, dass LLMs zum Aufdecken gängiger Sicherheitslücken verwendet werden können.

Diese Arbeit erweitert das Wissen im Fachgebiet, indem sie die Wirksamkeit von LLM-gesteuerten Tests für REST-APIs hervorhebt und Strategien zur Optimierung der LLM-Leistung durch fortschrittliche Prompt-Techniken anbietet.

Abstract

This thesis investigats the efficacy of Large Language Models (LLMs) in automating REST API test case generation. It focuses on their performance, their ability to identify vulnerabilities and improve test coverage and how Prompt Engineering techniques effect this. REST APIs are crucial in web development due to their simplicity, but they are vulnerable to attacks. Therefore, thorough testing is essential. LLMs can help with that as they improve safety and resilience because they can help streamline testing and documentation procedures for APIs. Thus, the goal of this thesis is to develop an LLM-based approach to generate test cases, improve coverage, and reduce expenses.

This work introduces two new use cases: WebAPIDocumentation and WebAPITesting. It aims to automate the generation of OpenAPI specifications, correct documentation inconsistencies, and conduct tests to identify vulnerabilities such as faulty authentication. The results show that GPT-4o-mini o1 is more effective at recognizing endpoints and parameters, particularly when prompts are enhanced with Prompt Engineering techniques like Tree-of-Thought(TOT), Chain-of-Thought (COT) and In-Context Learning (ICL). Even though GPT-4o-mini does well in detecting common vulnerabilities like Identification and Authentication Failures and NoSQL Injection, it has difficulties with more complex issues such as Sensitive Data Exposure and Insufficient Rate Limiting. The results show the potential of LLMs to improve security testing and highlight the role of Prompt Engineering in improving the efficiency of automated testing. Overall, this thesis contributes to the current field by showing that LLMs can be used in generating test cases for REST APIs and are efficient in doing so. Their performance can be improved by using Prompt Engineering strategies, such as COT. Additionally, the work shows that LLMs can be used to find common security vulnerabilities.

This work contributes to the field by showcasing the effectiveness of LLM-driven testing for REST APIs and providing strategies to optimize LLM performance by using Prompt Engineering techniques.

Contents

CHAPTER 1
Introduction

1.1 Motivation

RESTful APIs (Representational State Transfer APIs) [Fie00] play an important role in modern web development because they are easy to implement, flexible, and scalable. Since they are not tied to a specific format, they are very flexible. However, they adhere to widely accepted web standards such as HTTP, JSON and XML formats, which allows them to make communication and data exchange between different applications that use different formats seamless. They are scalable because they use a stateless architecture. Therefore, they are important for a wide range of services, ranging from cloud computing to mobile applications [Ibm24, AM23].

Thorough testing of REST APIs is important to ensure proper functionality, effective error handling, and smooth integration with other systems. This is essential because even small API errors can spread through linked services and potentially cause widespread failures [MLSRC22]. Furthermore, thorough testing improves the user experience by ensuring that APIs can effectively manage real-world traffic. [GZA23, AM23].

Moreover, REST API testing is not only important to ensure proper functionality, but also security. This is because REST APIs often expose sensitive information like financial information or critical functionalities, which makes them appealing targets for attacks. Thorough testing in this situation helps in finding possible weaknesses like Injection attacks. Additionally, it ensures the confidentiality and integrity of the systems by protecting them from potential cyberthreats. Ineffective testing can expose APIs to exploitation, which can have serious consequences such as financial losses and data breaches. As a result, robust testing is a crucial part of API development and deployment since it improves performance while strengthening the overall security posture of web services [GZA23, Arc19b, AM23, EACM22].

In conclusion, using thorough testing strategies, can help organisations to build more robust REST APIs. It can not only ensure that they meet functional requirements but also withstand potential security threats. This can lead to greater trust and reliability in the services they provide [GZA23, AM23].

1.2 Problem Statement

Many current REST API testing tools like EvoMaster heavily rely on OpenAPI specifications [Arc19b, AGP19]. The problem is that these specs are frequently out of date or incomplete. When a specification is incomplete or outdated, the REST API testing tools can struggle to properly test APIs, which can lead to bugs or integration issues [GZA23, DPP+24]. In addition, most tools are not built to handle the complex structure and interdependencies of real-world APIs. This makes it usually hard for them to generate comprehensive test cases. As a result, developers often have to step in and manually fill the gaps to make sure edge cases and less obvious scenarios are properly tested [GZA23].

Even though, automated test case generation offers a promising solution to these problems. The existing methods struggle with complex data types and functionalities of REST APIs. Therefore, they can miss important unstructured data from API documentation, which limits their effectiveness. Another issue is that it is hard to compare these tools [KXSO22, MLSRC22].

The use of Large Language Models (LLMs) can help overcome these limitations since they can improve the automation and effectiveness of generating test cases. They can do this because they can better adjust to different API architectures and collect valuable insights from unstructured data. However, there are also disadvantages using LLMs. A common problem with LLMs is that they often struggle to generate the correct inputs and fully understand the technical details of APIs, which can lead to errors. This happens when APIs behave differently across versions or settings [KCS+23, JLM+24].

There are two main reasons for this. First, APIs vary widely in their design, which makes it difficult for LLMs to interact with them consistently. Since each API has a different structure, LLMs must continuously adapt, which adds complexity to the testing process. Second, LLMs have not been completely incorporated into the tools and workflows that developers typically use for API testing [DPP+24, JLM+24, AM23, EACM22] such as Postman. As a result, these models are still trained mainly on general-purpose text, not the kinds of logs or structured data that reflect real-world testing needs. This makes it harder for them to respond effectively when an API behaves in unexpected ways [QLY+24]. Moreover, since APIs frequently change (e.g., updates, new versions, different configurations), and without proper integration, LLMs struggle to keep up with these variations. In addition, most API testing tools (such as Postman or JMeter) are not yet designed to seamlessly work with LLMs [MLSRC22]. This makes it harder for LLMs to analyze API responses effectively, learn from failures, or suggest improvements [KSO23, JLM+24].

The lack of standardized practices in API design makes these challenges even more difficult. When APIs do not follow consistent design conventions, automating the testing process becomes much harder [KSO23]. To improve the automation of REST API testing, it is important to address these issues. Since it will help to ensure the reliability and performance of web services and applications that depend on these APIs. One key step toward a more efficient and resilient testing system is to use advanced techniques like LLMs, which can better understand complex commands and process unstructured data. This will help meet the growing demands of modern web development [KSO23, KXSO22, DPP$^+$24, AM23].

1.3 Research Questions

This thesis investigates the following research questions regarding the use of Large Language Models (LLMs) in testing REST APIs:

- **RQ1:** What is the efficacy of various Large Language Models (LLMs) in generating test cases for REST APIs, and which factors significantly influence their performance?
- **RQ2:** What prompt engineering techniques are most effective for generating test cases for REST APIs using LLMs?
- **RQ3:** To what extent can LLMs identify potential failure points or vulnerabilities within REST APIs, and how accurate are their predictions in this context?

1.4 Aim of the Thesis and Expected Results

The aim of this thesis to use Large Language Models (LLMs) to help automate the documentation and testing of REST APIs. The goal is to create an approach that automatically generates tests using LLMs that can handle many tricky aspects of testing REST APIs 3. These challenges include [GZA23, MLSRC22]:

- **Challenges in Standardization and Documentation:** The inconsistent use of standards like the Zalando[1] or Microsoft[2] REST guidelines creates problems such as non-standard API designs, outdated documentation, and challenges in integrating services, which makes it harder to maintain accurate documentation and resulting in incomplete or incorrect test cases. Our solution is WebAPIDocumentation, which automatically creates a standardized OpenAPI specification. This ensures that the specification stays up-to-date by creating a new version whenever the REST API changes. By following established REST guidelines like Microsoft and Zalando, it helps maintain a consistent API design, keeps the documentation

[1]https://opensource.zalando.com/restful-api-guidelines/
[2]https://github.com/microsoft/api-guidelines

current, and improves test case accuracy by providing a reliable source of truth for API behavior [EACM22].

- **Complex Input Types:** Dealing with complex input types and data structures in RESTful APIs can make it difficult to infer test input values accurately, limiting the scope of automatic test case generation to simpler data types. Since WebAPIDocumentation uses GET requests to query the REST API and retrieve the majority of data structures, it helps with inferring valid test input values. When required, it can handle complex data types or missing fields by using extra methods like schema analysis and API documentation parsing, extracting various input variations, and analyzing multiple responses.

- **Security concerns:** Ensuring the security of APIs while allowing effective testing is a significant challenge, as testing mechanisms often need to bypass or simulate security protocols, which can lead to vulnerabilities if not handled carefully. We address this challenge with WebAPITesting, which looks for common security risks, including those in the OWASP Top Ten, in REST APIs. It also detects API-specific issues like improper authorization and data exposure, ensuring security protocols remain intact during testing [Lim24, EACM22].

- **Automatic Generation of multistep test cases:** Testing REST APIs requires building complex sequences of API calls to mimic real-world situations, focusing not just on understanding each API function, but also on how they work together across different steps, essential for guaranteeing that the system behaves as expected by accurately simulating these sequences under various conditions. Since WebAPITesting look for security vulnerabilities, it automatically generates and executes multistep test cases to simulate real-world API interactions. It analyzes API dependencies to build accurate call sequences, ensuring both functional correctness and security under various conditions [CP08, MLSRC22].

By automating the generation of test cases with an LLM, the thesis aims to enhance the effectiveness and efficiency of REST API testing. This automation is expected to tackle the complex challenges as described above. Through this, the thesis will offer a new way of testing that can adjust to changes in APIs over time. This improves the safety, security, and robustness of tested web services while also reducing test generation costs and increasing coverage.

CHAPTER 2 Background

In this section, we are going to give a short overview of RESTful APIs, how they are tested, and what are the challenges in testing.

2.1 REST API Testing

RESTful APIs (Representational State Transfer APIs) are web services that use the REST architectural style that was introduced by Roy Fielding in 2000 [Fie00]. This architecture helps create scalable web systems where the client and server use stateless communication. This means that every client request has all the data needed to process it and no client context is stored on the server. Moreover, RESTful APIs are flexible in how data is exchanged because they do not need a specific format. Therefore, applications, which are built using different technologies, can communicate with each other since REST APIs can be used with different formats such as JSON and XML [GZA23, AM23, Ibm24].

A key advantage of RESTful APIs is that they use **HTTP**(Hypertext Transfer Protocol) methods to interact with resources at specified endpoints [RFBL99, EACM22]. The methods are POST, GET, DELETE, PUT and PATCH, where GET is the only one that does not change the state of the resource. The resources are identified by a unique URI (Unique Resource Identifier) and can be texts, photos, or representations of actual objects. When a resource is queried, the client sends an HTTP request containing an url to server, which returns the resource in formats such as JSON, HTML, or plain text [GZA23]. For instance, if the request " GET /users/123" is made, then this means that there is a GET request for the resource user with ID 123 and if there exists a JSON resource of this user, then ""id":"123"" is returned [GZA23, Ibm24].

Another advantage of REST APIs is that they handle each request independently, which removes the need for stored session data in the system. Therefore, REST APIs become easier to scale and more dependable as a result. Additionally, responses that are listed as "cacheable" can be used to lessen the amount of times a server is called and enhance performance [GZA23, RFBL99, FM24].

A common standard for REST API documentation is **OpenAPI Specification (OAS)** and offers a language-neutral format that makes it possible for machines and people to understand how to use an API without having to see the source code. OpenAPI supports automated tools that can generate easy-to-read documentation, server code (stubs), client libraries, and testing scripts. OpenAPI documents are usually written in JSON or YAML and describe how to use API endpoints by specifying details like URIs, HTTP methods, and input parameters. Fuzzing tools also often use it to test APIs for security issues by showing what requests are allowed, what responses to expect, and which data formats are supported [GZA23, Tea, TTH21, ISCK24].

2.1.1 Testing RESTful APIs

REST APIs must be thoroughly tested to make sure that they work as intended, manage requests and data appropriately, and scale effectively in a variety of scenarios [RR07]. Moreover, testing can help catch issues like incorrect data processing, security vulnerabilities, and performance issues [Mye04]. To speed up this process, testing can be automated as it helps to simplify the testing process, provides thorough coverage, and ensures that the API performs reliably in various scenarios [GZA23, ISCK24].

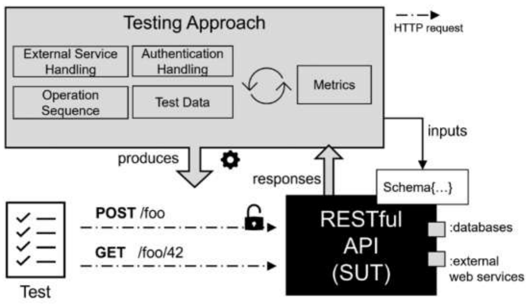


Figure 2.1: Overview of REST API Testing at a High Level taken from [GZA23]

RESTful APIs are usually tested using a structured approach. First, a schema is defined that specifies how the REST API should interact with the resources [Fie00, GZA23] (see Figure 2.1). For this purpose, OpenAPI specification or JSON are often used. Second, the schema is used to generate test cases that are evaluated against predefined metrics. Third, the testing tools execute sequences of HTTP requests, which are defined in the test case. Usually, these requests are POST or GET with specific parameters to evaluate the behavior of the REST API. Additionally, it is tested if the authentication mechanism of the REST API works correctly and it can connect properly to databases or other services. Developers often use smart strategies like heuristics to try to cover all possible scenarios [GZA23].

Approaches

REST API testing techniques can be categorized into different categories (see Table 2.1):

Category	Description	Tools
Black-box and White-box Testing	Black-box tests API behavior, while white-box looks at internal structure. Hybrid methods improve input handling [GZA23].	EvoMaster [Arc19b], QuickRest [KCS20]
Search-based Testing	Uses algorithms like Genetic and Swarm to optimize test generation and maximize coverage [GZA23].	EvoMaster [Arc19b]
Property-based Testing	Validates schema and resource state consistency [GZA23].	Schemathesis [HDD22], QuickRest [KCS20]
Model-based Testing	Uses models like Petri Nets are used to help the test generation [GZA23, FB15, LLZ18].	–
Others	Graph-based approaches, regression tools, and LLMs enhance test generation and fault detection [GZA23].	ExVivoMicroTest [GGM$^+$23, GGM$^+$20], LLMs [KSS$^+$24, BMR$^+$20, DPP$^+$24]

Table 2.1: Summary of REST API Testing Techniques

Black-box testing checks how the REST API behaves based on its inputs and outputs, without looking at the internal code. On the other hand, **white-box testing** looks at the code itself. Tools like EvoMaster [Arc19b] can be used for both black-box and white-box testing whereas QuickREST [KCS20] only works for white-box testing. A mix of black-box and white-box testing methods is usually used to manage complex input scenarios and improve the coverage and effectiveness of the test [MASR21].

A different method that is called **search-based testing**, uses metaheuristic algorithms like Swarm Algorithms and Genetic Algorithms to improve test case generation optimization. Algorithms such as MOSA (Many-Objective Sorting Algorithm) [PKT15] and MIO (Many Independent Objective) [Arc19a] have been developed for testing REST APIs to improve how these tests are performed [GZA23]. In contrast, EvoMaster combines these search-based techniques to optimize code coverage during testing [Arc19b]. In addition to these, there's property-based testing, which ensures certain conditions are consistently maintained throughout the testing process, such as making sure the state of resources and the structure of the data remain correct. Tools like Schemathesis [HDD22] and QuickRest [KCS20] verify the structural and semantic correctness of REST APIs [GZA23].

Model-based testing takes a different approach by using abstract representations to simulate the API's behavior and state transitions. Models such as UML state machines or colored Petri Nets serve as blueprints to guide the generation and verification of test cases [LLZ18, FB15]. Besides these methods, other testing approaches are also used. For example, **graph-based testing** uses algorithms like Breadth-First Search (BFS) to explore how different parts of a REST API are connected, which helps to create more complete test cases. Regression testing tools like ExVivoMicroTest compare different versions of an API to find bugs that may have been introduced during updates [GGM$^+$23, GGM$^+$20, GZA23].

Since Large Language Models (LLMs) can create test cases from natural language descriptions of API features, their integration has grown in popularity recently. LLMs can improve test coverage and efficiency by generating input, predicting output, and inferring test scenarios based on API documentation [KSS+24, BMR+20].

In conclusion, the testing approaches covered above have different advantages in automation, optimization, or modeling. If these approaches are used together, this could improve testing effectiveness and ensure reliable REST API performance.

Metrics

To assess how effective testing is, three main categories of metrics are commonly used: coverage criteria, fault detection, and performance (see Table 2.2) [GZA23]. Each category is important to ensure that APIs work as intended, adhere to specified schemas, handle faults effectively, and maintain performance under different conditions.

Category	Description	Tools
Coverage Criteria	**Schema coverage:** Focuses on status codes, paths, and responses.	HsuanFuzz [TTH21],
	Code coverage: Measures executed code.	RESTest [MSR21],
	Specialized metrics: Inter-parameter dependencies and security analysis.	JaCoCo [BT24],
		EvoMaster [Arc19b]
Fault Detection	**Service errors:** Detected using HTTP 5xx codes.	RestTestGen [CZPC22],
	Schema violations: Deviations from API schema.	QuickRest
	Rule violations: Idempotency, security, and consistency.	
Performance	**Response times:** Evaluated with testing strategies [GZA23, BFGC21].	Apache JMeter [FB15],
	Workloads: Simulated with Apache JMeter [FB15].	Schemathesis [HDD22]
	Bottlenecks: Identified by Schemathesis [HDD22].	
	Latency: Analyzed across regions [BIEC20].	

Table 2.2: API Testing Metrics

Coverage criteria help measure how thoroughly an API has been tested by looking at different parts of its behavior. One key metric is *schema-related coverage*, which checks whether the API follows the structure defined in specs like OpenAPI including expected inputs, outputs, status codes, and available operations. Testing tools of this type are HsuanFuzz [TTH21] and RESTest [MSR21], which help test these elements by generating requests and checking if responses match the schema. Another common metric is *code coverage*, which measures how much of the backend code is actually executed during testing. Tools of this type are EvoMaster [Arc19b] and JaCoCo[1], which help track this and is especially useful for Java and JavaScript applications. Some tools go further by using *special metrics* to examine how input parameters interact or how secure the API is. For example, grammar-based coverage looks at how components relate to each other, while NLP-based tools can create realistic input data to mimic real-world users and scenarios.

Fault detection identifies *errors*, *schema violations*, and *rule violations*. First, service errors are often shown through HTTP status codes: 5xx codes signal server issues, while 4xx

[1]https://github.com/jacoco/jacoco

codes point to client errors, such as bad requests or authentication failures [RR07, FT02]. Second, schema violations usually occur when an API response does not match what the schema defines. Tools of this type are RestTestGen and QuickRest. Third, rule violations involve breaking security policies or failing to meet expected behaviors, such as idempotency (consistent results from repeated requests). Another method is **metamorphic testing**, which checks for errors across multiple executions. Lastly, **regression testing** catches issues introduced by changes in new versions of the API [GZA23].

Performance measures how well an API handles requests under different conditions. Banias et al.[BFGC21] analyzed *response times* to find methods that reduce delays. Fertig and Braun[FB15] used Apache JMeter to test how the API performs under different *workloads* and network conditions. Schemathesis [HDD22] looked for bottlenecks by identifying slow responses and excessive requests, while Bucaille et al. [BIEC20] developed a system to track *latency* and performance in multiple locations, highlighting how infrastructure and geographic factors affect performance.

In conclusion, the effectiveness of the REST API testing is evaluated using coverage criteria, fault detection, and performance metrics. The coverage criteria ensures how thorough the testing was, while fault detection identifies errors and violations. Performance evaluates response times and bottlenecks. Tools such as RestTestGen, Apache JMeter, and Schemathesis support these analyses.

2.1.2 Challenges in Automating REST API Tests

Testing REST APIs presents several challenges that can complicate the process and reduce test effectiveness (see Table 2.3) [KXSO22]:

Category	**Description**
External Dependencies	Since REST APIs often rely on external services, setting up a stable testing environment can be challenging [EACM22, CP08].
Schema Evolution and Standardization	When REST APIs evolve and standards vary, then inconsistencies in design and outdated documentation become issues [EACM22].
Complex Input and Multi-Step Test Generation	It is difficult to replicate real-world scenarios, especially when REST API calls depend on each other. [KXSO22, EACM22]
Security and Authentication	Testing secure APIs requires simulating protocols like OAuth without exposing sensitive information [Tec24, EACM22].
Fault Detection and Dynamic Data	External dependencies and changing data make it harder to detect faults and generate automated tests [CP08, MLSRC22].
Performance Testing	Testing response times often requires advanced tools and the ability to simulate real-world user loads, which can generate costs [CP08].

Table 2.3: Challenges in API Testing

Testing REST APIs has several challenges that can make the process complex and reduce the accuracy of test results. One big challenge is external dependencies because REST APIs often rely on other services, like databases or third-party APIs, which may not always be available or return consistent results during testing [EACM22]. Another challenge is **the evolution and standardization of the scheme**. This happens, when a REST API changes because a new endpoints is added or an existing endpoint is modified, it is hard to keep the tests up to date. The problem gets worse when developers do not follow common standards like those from Zalando or Microsoft. This can lead to nonstandard designs and outdated documentation [Tec24, KXSO22, EACM22].

Handling **complex input and multistep test generation** is also challenging, especially when APIs require sequences of interdependent calls to mimic real-world scenarios. Generating such tests becomes even harder when dealing with complex data types or ensuring that requests depend on the correct sequence of prior responses. Moreover, most tool have been tested in controlled environments, which makes hard to know how they will perform in the real world [MLSRC22, KXSO22, Lim24].

Security and authentication introduce additional complications, as testing secure APIs requires bypassing or simulating security mechanisms such as OAuth or API keys without compromising security measures [MLSRC22, Tec24, EACM22].

An additional challenge is **identifying faults and handling dynamic data**. Pinpointing the source of problems, such as 5xx server errors, can be difficult when it is uncertain whether the issue originates from the API or its external components. This becomes even more complicated with dynamic data, such as timestamps or auto-generated IDs, which make it difficult to create reproducible automated tests [MLSRC22, Tec24].

Lastly, **performance testing** poses its own set of difficulties. Accurately measuring response time, latency, and throughput requires sophisticated infrastructure and realistic simulations of user load, which can be especially demanding for distributed systems that need to handle high levels of concurrent requests [MLSRC22, KXSO22, Lim24].

In conclusion, there are challenges in automating REST API testing. These are include dealing with external dependencies, keeping up with changing schemas, and creating complex, multistep test scenarios. Moreover, it is important to tackle issues such as authentication, handling dynamic data, and measuring performance. Addressing these challenges calls for strong tools and infrastructure to ensure tests remain accurate and reliable, even in constantly changing environments.

2.1.3 Summary

Modern web services depend on RESTful APIs because they enable stateless and scalable client-server communication using common HTTP methods like POST, GET, DELETE, PUT, and PATCH. Schemas like OpenAPI ensure consistent interactions, simplify testing, and support integration. Testing APIs is important to verify functionality, reliability, and scalability (see Figure 2.1).

Table 2.1 shows different API testing methods. Black-box testing evaluated API behavior without accessing internal details, whereas white-box testing examines internal structures. Using a combination of both techniques achieves better results and is called hybrid testsing. Other methods are search-based algorithms and model-driven testing that optimize test generation. More recent innovations such as graph-based techniques and Large Language Models (LLMs) further improve testing.

API testing metrics (Table 2.2) include coverage, fault detection, and performance. Coverage ensures thorough tests by checking schema compliance and parameter interactions. Fault detection identifies errors such as schema mismatches, while performance metrics measure response times and locate bottlenecks. Tools such as EvoMaster and Apache JMeter aid in evaluations.

Automating API testing faces challenges (Table 2.3), such as adapting to schema changes, simulating interactions, and ensuring security for features such as OAuth. Effective solutions require robust tools and strategies to handle these complexities.

This thesis uses black-box testing as the main approach because it evaluates an API's behavior without needing access to its internal structure, making it suitable for real-world use. Metrics such as routes and parameters discovered, vulnerabilities found, execution time, and workload analysis offer a thorough assessment of automated REST API testing. These metrics help identify issues, measure performance, and address common challenges, supporting the creation of effective solutions for testing APIs in complex environments.

2.2 Large Language Model

Large Language Models (LLMs) are a type of machine learning model that are designed to comprehend and produce text that is similar to that of a human. They use transformer-based architectures like the models Bidirectional Encoder Representations(BERT) [DCLT19] and the new Generative Pre-trained Transformer (GPT) [Suf24], which are deep learning methods. Usually these models are trained on large datasets, which enable them to produce and predict text based on context. Moreover, accurate language generation and understanding are influenced by quality of the training data.

Year	Name	Developer	Description
2018	**BERT**	Google	Introduced bidirectional attention for better language understanding. [DCLT19]
2019	**GPT-2**	OpenAI	Generated fluent text with 1.5B parameters. [Suf24]
2020	**GPT-3**	OpenAI	Expanded to 175B parameters, enabling more coherent and general-purpose text generation. [BMR+20]
2021	**Gopher**, **Chinchilla**	DeepMind	Focused on efficient scaling by balancing model size and training data. [RBC+21, HBM+22]
2022	**BLOOM**, **OPT-175B**	Meta AI	BLOOM emphasized openness; OPT-175B rivaled GPT-3 in scale. [SFA+22]
2023	**GPT-4**	OpenAI	Improved reasoning and context handling, estimated to use 1T parameters. [Ope23]
2023	**LLaMA**	Meta	Provided smaller, efficient models; LLaMA2 improved fine-tuning. [TLI+23, TMS+23]
2024	**GPT-4o**, **GPT-o1**, **GPT-o1-mini**	OpenAI	Lighter GPT-4 versions tailored for speed, cost, and performance. [Ope24c, Ope24a]
2023–2024	**Recent Developments**	–	Models like Falcon, Mistral 7B, and Code Llama advanced coding and summarization. [JSM+23, RGG+23]

Table 2.4: Important Milestones in the Development of Large Language Models

Table 2.4 shows how LLMs have evolved over time. Early models like BERT (2018) [DCLT19] introduced bidirectional attention, which improved how machines understand natural language. GPT-2 (2019) [SWR+22] and GPT-3 (2020) [BMR+20] demonstrated the power of scaling up model size, with GPT-3 making a big impact by generating coherent text using 175 billion parameters [Suf24, Ope23]. To overcome the limitations of earlier models, DeepMind developed Gopher [RBC+21] and Chinchilla (2021) [HBM+22], which optimized the balance between data size and the number of parameters.

LLMs became more widely available in 2022 when BLOOM [SFA+22] and OPT-175B [ZRG+22] released open-source models. A year lated, GPT-4 [Ope23] trillion parameters was released and produced more coherent text and improved context understanding then the models before. In contrast, Meta's LLaMA [TLI+23] and LLaMA2 [TMS+23] presented more compact, effective models that were better suited for fine-tuning. Nowadays, more sophisticated models like Falcon-180B, Mistral [JSM+23], and Code Llama (2023-2024) [RGG+23] can manage intricate tasks like coding, summarizing data, and having conversations.

GPT-4o [Ope24b], GPT-4o mini, and o1 [Ope24c] were added to OpenAI's language model lineup in 2024; each was created to meet a particular need. Launched in May, the GPT-4o became the flagship model, able to process text, images, and audio simultaneously for multi-modal, real-time tasks. Two months later, the more affordable, and compact GPT-4o mini was released. A bigger model, o1 model, was unveiled in September, which is good for tasks like logical analysis, multi-step reasoning, and research. This is because it emphasizes sophisticated reasoning and complex problem-solving. Overall, these models show that OpenAI is trying to find a balance between accessibility, efficiency, and power across a range of applications.

In summary, LLMs are a major advancement in natural language processing (NLP), which makes it possible for machines to understand and generate text like humans.

2.2.1 Functionality

The **Transformer architecture** is a deep learning method used by LLMs to understand and generate text that seems to have been written by a human. It was originally presented by Vaswani et al. (2017) [VSP+17], which has changed the way that machines process language by allowing models to deal with complex linguistic structures and "long-range dependencies" accurately.

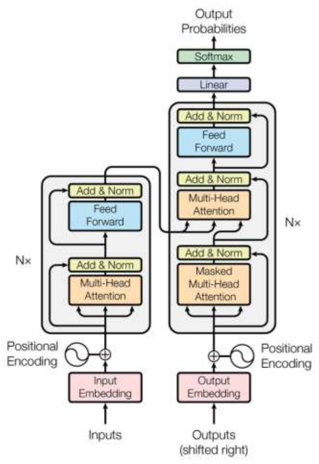


Figure 2.2: Transformer architecture taken from [VSP+17]

A **Transformer** architecture is divided into an **encoder** and a **decoder**. These components two work together to process input text and produce logical output sequences. The original text is transformed into numerical formats known as **embeddings**, as illustrated in Figure 2.2. The addition of **Positional Encoding** preserves the sentence's structure and helps the model in comprehending the word order, which is crucial for accurately capturing syntax and meaning [VSP+17].

The **encoder** plays an important role in analyzing the input using a mechanism called **self-attention**. It helps the model to understand how words relate to each other, even when they are far apart in the sentence [VSP+17]. For example, in *The cat, which I saw yesterday, is fluffy*, the model has to understand that *fluffy* describes *cat*, even with other words in between. This ability to maintain context over long distances is what makes LLMs built on Transformers so powerful.

A **Transformer** is able to solve difficult tasks due to two properties. First, transformers are able to better understand the word order in a sentence because of **Positional Encoding**. If they would not have this encoding, transformers would not be recognize the word sequence because they view the entire sentence at once. They keep track of each word's location by giving it a number [VSP+17]. Without this, a sentence could be unclear, such as "The cat, which I saw yesterday, is fluffy," where the model does not know, which words go together.

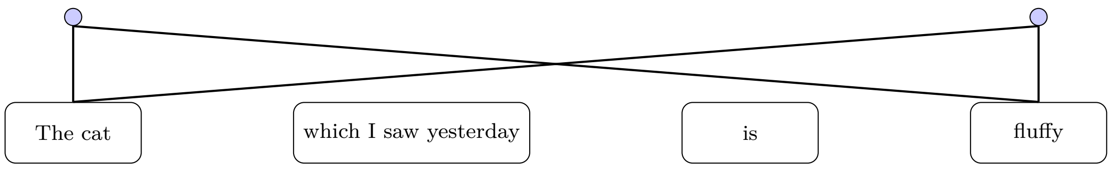


Figure 2.3: Self-attention links "cat" and "fluffy," ensuring the model captures their relationship.

Second, as already mentioned, **self-attention** enables the model to examine all words in a sentence to determine which are the most relevant to each other in understanding context. Unlike older methods that focus on nearby words, self-attention assigns importance, or "attention weights," to words based on their relationships, even over long distances [VSP+17]. In "The cat, which I saw yesterday, is fluffy," for example, the term "cat" is very different from "fluffy." To make sure that the meaning of the sentence is accurately captured, self-attention helps the model identify that "fluffy" describes "cat" by giving this connection more weight (see Figure 2.3). The model can successfully understand both local and long-distance word relationships thanks to this mechanism [VSP+17].

The **decoder** uses the encoded input together with the generated words from before to predict the next word. During training, it learns to align the generated text with the input using the internal attention mechanisms. A loss function reduces prediction errors to increase accuracy, and the output is transformed back into text using *output embeddings*. After processing the input, the decoder uses a linear layer and a softmax function (see Figure 2.2) to turn its outputs into "token probabilities", which makes it possible to generate coherent text [VSP+17].

In conclusion, the **Transformer architecture** is a fundamental component of current **LLMs**, which enables these models to excel in comprehending and producing text by efficiently capturing structure, meaning, and relationships within language. This is achieved because the **positional encoding** and **self-attention** are combined.

Training LLMs

In comparison to more traditional models like RNNs that process input step by step, transformers make it possible for LLMs to process entire text sequences at once, which makes them faster and more efficient. Once these models have learned general language patterns from large and diverse datasets, they can summarize, translate, and converse [BMR+20]. Fine-tuning [HR18, RSR+20] helps them adapt to particular tasks or applications, which further improves their performance.

The LLM training process can be summarized as follows [KAT+25]:

1. **Pre-training:** The first step of training a LLM is called *pre-training*, where a model is trained on datasets from various text sources like books and websites. Then it learns general language patterns by doing word prediction or filling in the blanks. This category of learning is called *supervised learning*, which helps the model, for instance, to understand syntax, semantics, and factual knowledge [KAT+25].

2. **Fine-tuning:** After the first step is finished, the model is further trained in a smaller and more specific-task set to adapt it to particular goals. For example, it can be fine-tuned for particular tasks like answering questions, summarizing, and sentiment analysis [RWC+19]. Some fine-tuning approaches are [KAT+25]:

    - **Transfer Learning**: The general knowledge learned in the pre-training step is fine-tuned to specific tasks using relevant data [VSP+17, DCLT19, HR18]. This helps to improve performance on downstream tasks [RSR+20].
    - **Instruction Tuning**: Instruction-formatted datasets like input-output pairs and task instructions are used to improve the models' zero shot generalization and task-specific performance [WBZ+22, SWR+22]. This enables models to process complex natural language commands effectively.

    Alignment: To ensure that LLMs are helpful, honest, and harmless (HHH) [ABC+21], alignment was introduced because it uses feedback to modify the behavior of the model and prevent it from producing inaccurate text [BJN+22]. One important alignment methods is *Reinforcement Learning with Human Feedback (RLHF)* [CLB+17, Ole25]. It works like the following:

a) **Reward Modeling (RM)**: A reward model is trained on a "preference dataset", which ranks responses based on human preferences and can be labeled with HHH criteria [Ole25, ABC+21].

b) **Learning Reinforcement (RL)**: The trained model is used to further refine the LLMs. Usually the optimizer used to adjust the model's parameters to improve performance is proximal policy optimization (PPO) [SWD+17] because it only makes small changes to the policy during training.

3. **Optimization and scaling:** Increasing the number of parameters can improve performance on a variety of tasks, including the ability of the model to capture complex language structures, according to research by Brown et al. [BMR+20]. For example, GPT-3 performs very well and it has 175 billion parameters [KMH+20]. However, just increasing number of parameters is not the best course of action. Research such as *Training Compute-Optimal Large Language Models* [HBM+22, ZLM+23] shows that better results can be obtained when model size, training data, and computational efficiency are balanced. Specifically, smaller models that have been trained with more data can occasionally perform better than larger ones [HBM+22, R24]. An example for such a model is Mistral7 [JSM+23].

In conclusion, one of the biggest advantages of LLMs is that they can produce text that stays coherent and context-aware. The broad patterns they learnt during their pre-training period are what give them this capacity. Moreover, LLMs can handle tasks with little or no examples because of their few-shot and zero-shot learning abilities [BMR+20]. Interestingly, research on inverse scaling [MLP+23] shows that bigger models do not always perform better, especially on tasks that involves complex reasoning. This suggests that scaling alone isn't enough. Instead, the quality of the model is heavily influenced by things like training data, architecture, and how efficiently resources are used.

2.2.2 Challenges in LLMs

Although LLMs have shown their abilities, they still face a number of challenges, which must be addressed o make sure they are suitable for real-life applications. These challenges include ethical issues, technical limitations, and practical constraints tied to the deployment of large-scale models (see Table 2.5).

One major challenge is **Bias and Ethics** because LLMs are trained on historical datasets. This sets that include human-written text with biases of the time, they inherit the biases, which exist in the training data. These biases can reflect gender, racial, cultural, or other social biases, leading to biased or inappropriate results. For example, an

Category	Description
Bias and Ethics	LLMs can pick up harmful biases from their training data, which may lead to unfair results in areas like hiring or healthcare [BGMS21, WMR+21].
Hallucinations	LLMs sometimes come up with wrong answers that sound reasonable, which can be dangerous when used in sensitive areas [JLF+23].
Computational Resources	Training and running LLMs uses a lot of computing power and produces lots of CO_2 emissions, which raises concerns about cost and their environmental impact [SGM19, BGMS21].
Interpretability	It is hard to understand how LLMs come up with decisions and this lack of clarity makes them harder to trust in important areas like law or medicine [DVK17, SIG+24].
Context Length Limitation	LLMs can only look at a limited amount of text at once, which makes it tough for them to handle long conversations or documents [BMR+20].

Table 2.5: Challenges in LLMs

LLM can generate discriminatory language or reinforce stereotypes if trained on biased data [BGMS21]. Although there are ongoing efforts to reduce this bias, it is challenging because much of the training data come from the vast and unfiltered content available on the Internet. This can become a problem because LLMs are being used more frequently in sensitive areas such as employment, law enforcement, and healthcare [WMR+21]. One major issue is that is known as hallucination. This happens when the model outputs text that sounds convincing but is actually false or made-up [JLF+23].This happens when LLMs does not reallly understand facts and they produce responses based on patterns in the data. As a result, they might sometimes create details or answers that seem accurate but are actually misleading [JLF+23]. This problem becomes particularly concerning when LLMs are used in applications such as healthcare care, testing, or any domain that requires factual accuracy, as hallucinations can result in harmful or dangerous outcomes. Therefore, it is important that the testing tools generate a report so that the testers can check and reproduce it to ensure the validity of the test.

Large amounts of **Computational Resources** are needed for LLMs. Especially, large-scale models like GPT-3 and o1 require a lot of processing power to train and run since they have billions of parameters and use potent GPU or TPU clusters to process massive datasets. This energy-heavy process is an environmental concern [SGM19, BGMS21] because training procedure emits a lot of CO_2. Moreover, due to the high computational costs it is harder for smaller organisations to get access , which concentrate AI development among a small number of well-funded institutions. Even after training, these models are still expensive to deploy for real-time use because of their high memory and processing requirements [SGM19].

Another issue with LLMs is **Interpretability**. Even though these models are accurate, it is challenging to understand how they arrive at particular conclusions because they work like "black boxes". This lack of transparency is particularly troubling in sensitive areas such as the medical field, where trust and rationale are essential. There is currently work is ongoing to improve interpretability, but definitive solutions have not yet been reached [DVK17, SIG+24].

Although LLMs can understand and produce human-like text, their ability to handle long-term dependencies is still **limited by their context window.** Models like GPT-3 have a finite context window, which means they struggle to capture very distant word connections in text [BMR+20]. This limitation becomes a problem for tasks that need to understand long documents or keep track of context in extended conversations.

In conclusion, although LLMs are great at producing natural-sounding language, they still have major challenges, such as built-in biases, errors, high computational requirements, limited transparency, and struggles with managing long-term information. Addressing these issues is crucial, especially in sensitive applications, and requires ongoing research to ensure responsible AI use.

2.2.3 Practical Applications

LLMs have significantly changed the way software developers write, fix, and improve code. Their ability to process and generate text has simplified the coding processes since it saves developers time and effort, as illustrated in Table 2.6.

Category	**Description**	**Tools**
Code Generation	Generates and completes code snippets based on prompts, which reduces repetitive tasks and improves productivity.	GitHub Copilot[2], OpenAI Codex[3]
Bug Detection and Debugging	Identifies coding errors and suggests fixes to streamline debugging and improve code reliability.	DeepCode[4], CodeWhisperer[5]
Code Review and Optimization	Provides suggestions to improve code efficiency, readability, and maintainability.	Codiga[6], TabNine[7]
Knowledge Assistance	Answers programming questions and explains technical concepts or APIs to assist developers.	ChatGPT[8], StackOverflow Assistant[9]
Testing Automation	Generates test cases and identifies edge cases to ensure robust and reliable software performance.	EvoMaster[10], Diffblue Cover[11]

Table 2.6: Applications of LLMs in Software Development

The automation of code creation is one important application of LLMs in programming as it makes it possible to save time . GitHub Copilot and OpenAI Codex are two examples of tools that let developers quickly build and test their ideas by converting typed prompts into functioning code [CTJ+21]. As a result, developers can focus on tackling more complex problems and cut down on time spent on monotonous work. However, the code must still be manually tested to ensure semantic accuracy even if the code is syntactically correct.

Additionally, LLMs can help with debugging by reviewing code, finding errors, and suggesting fixes. There are tools like CodeWhisperer and DeepCode make the debugging process faster and easier. Other tools, such as Codiga and TabNine, help improve code quality by making it more readable, maintainable, and easier to review and optimize.

Furthermore, LLMs provide extremely helpful knowledge support by **answering programming queries, explaining complex ideas, and providing documentation**

for APIs and libraries. For example, ChatGPT makes it easier for developers to learn and troubleshoot. To ensure reliable program behavior, LLMs also automate the creation of test cases and identify edge cases using tools like EvoMaster and Diffblue Cover.

In conclusion, LLMs have transformed software development by speeding up coding, debugging, testing, and code reviews. Tools like Codex and Copilot can help automate routine coding, testing, and debugging tasks (see Table 2.6).

2.2.4 Summary

Large Language Models (LLMs) use the Transformer architecture, which helps them to understand and generate human-like language. Popular models are like BERT [DCLT19], GPT-3 [BMR+20], GPT-4 [Ope23], and LLaMA [TLI+23]. If one wants to use the model for a specific purpose, it has to be fine-tuned. These models are particularly good at understanding word relationships and context using techniques like self-attention and positional encoding.

This thesis compares GPT-4o-mini and GPT-o1, which represent different sizes and levels of optimization in LLMs. GPT-4o-mini is designed for high-performance tasks but is smaller and more affordable than bigger models. On the other hand, GPT-o1 and its smaller version, GPT-o1-mini, are built for speed and efficiency. This makes them a good fit for situations where fast responses and lower resource use are important. This thesis looks at what each model does well, where they struggle, and how they work in real-world cases. It also shows how model size and optimization affect performance in different situations.

While LLMs offer many advantages, they also come with some challenges. These include bias in the training data, generating incorrect but convincing answers (called "hallucinations"), high computing costs, and difficulty handling long-term context (see Table 2.5). Fixing these problems is especially important in areas like healthcare and law, where accuracy and fairness are critical.

LLMs are also becoming useful tools in software development. Tools like OpenAI Codex and GitHub Copilot can help automate tasks like coding, debugging, and testing (see Table 2.6). Other LLM tools can support code reviews, give programming tips, and generate test cases to save time and improve code quality.

In summary, LLMs are good at understanding and generating language. They represent a big step forward in AI. This thesis looks at how models like GPT-4, GPT-o1, and GPT-o1-mini deal with current challenges and how well they perform in real-world tasks.

2.3 Prompt Engineering

Prompt Engineering was created to make the answers from Large Language Models (LLMs) more accurate and relevant. Since LLMs heavily rely on the quality of the prompts they receive, well-structured prompts help reduce errors, ensure precise answers, and make interactions more efficient [SIB+24]. Prompt Engineering makes LLMs more efficient and useful by saving time, reducing costs, and allowing for customized interactions across industries. It has grown from basic prompts to more high-level methods like **Chain-of-Thought**, **Tree-of-Thought**, and **In-Context Learning**, which help LLMs handle complex reasoning. As a result, it has become a key research area for improving accuracy and relevance in AI responses [SIB+24].

2.3.1 Chain-of-Thought

The Chain-of-Thought (COT) walks the LLM through "reasoning steps" to help them tackle complicated problems [WWS+22]. This is done by dividing a task into smaller logical steps rather than giving an answer right away, which allows the model to "think out loud". This increases accuracy and simplifies the reasoning process, especially for multi-step tasks like logical reasoning, math problem solving, and common sense comprehension [WWS+22, SIB+24].

Using prompts such as "Let's solve this step by step" [KGR+22] helps COT the model to break the given prompt into smaller steps. It breaks down challenging tasks into smaller, more manageable sub-tasks, which prevents the model to rush its' responses thereby improving accuracy. Through an explanation of its reasoning, the model becomes more transparent, which reduces errors. For multi-step tasks like logical reasoning and math problems, COT is helpful as these tasks need a structured approach rather than an immediate answer. Moreover, it helps the model to produce more thoughtful and organized answers across different domains[WWS+22, SIB+24].

For example, in order to solve the question "What is the sum of the first 10 even numbers?" [Agg23], the phrase "Let's break this down step by step." [WWS+22, SIB+24] can be added before it. The model breaks the problem down into three steps. In the first step, the model lists the first ten even numbers (2 to 20), then it adds them up one by one ("$2+4 = 6$", "$6+6 = 12$", ...), and finally gives the correct answer, which is "110". In order to catch mistakes, a check like “If the Total is over 100, double-check your math” can be added [R24, KGR+22, WWS+22].

In short, COT helps the model solve complex problems by breaking them into smaller steps, leading to more accurate and relevant answers.

2.3.2 Tree-of-Thought

Instead of using a linear step-by-step process, the tree-of-thought (TOT) Prompt Engineering strategy incorporates branching paths of reasoning, building on the chain-of-thought (COT) approach [YYZ+23, SIB+24]. To arrive at a final answer, the TOT strategy allows the model to evaluate multiple potential solutions in a tree-like structure [YYZ+23]. This method is particularly useful for tasks like planning, problem solving, and scenarios that call for decision making in the face of uncertainty or where there are many different options to choose from [GOP+23, SIB+24, Lon23, LZ25].

An work that uses a similar approach is PentestingGPT [DLMV+24], which uses the Pentesting Task Tree (PTT) that resembles the structure of a TOT prompt (see Figure 2.4).

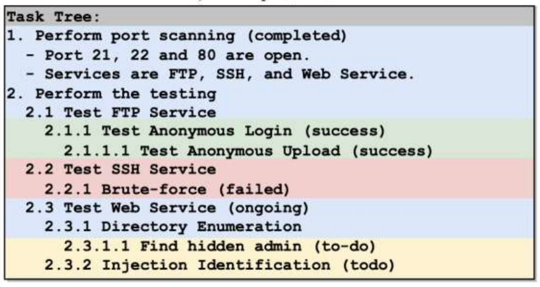

```
Task Tree:
1. Perform port scanning (completed)
  - Port 21, 22 and 80 are open.
  - Services are FTP, SSH, and Web Service.
2. Perform the testing
  2.1 Test FTP Service
    2.1.1 Test Anonymous Login (success)
      2.1.1.1 Test Anonymous Upload (success)
  2.2 Test SSH Service
    2.2.1 Brute-force (failed)
  2.3 Test Web Service (ongoing)
    2.3.1 Directory Enumeration
      2.3.1.1 Find hidden admin (to-do)
    2.3.2 Injection Identification (todo)
```

Figure 2.4: Pentesting Task Tree taken from [DLMV+24]

If one wants to solve the question "What is the sum of the first 10 even numbers?"[Agg23] in TOT, one has to add the following prompt proposed in [Hul23]:"Imagine three different experts are answering this question. All experts will write down 1 step of their thinking, then share it with the group. Then all experts will go on to the next step, etc. If any expert realises they're wrong at any point then they leave. The question is...". The experts represent three different branches, where each tries to find a solution. Each expert comes up with different ideas: one lists and added the numbers, another uses pairing, and the third applies the formula for sum of an arithmetic series. As a next step, they compare their reasoning step by step. At the end, they all arrived at the answer"110", which confirms each reasoning. No expert dropped out, as all paths were correct and reached the same answer.

By allowing the model to avoid unhelpful paths and focus on better options, the TOT approach makes problem-solving more flexible and increases the chances of finding the best answers to difficult questions. This branching method helps the model explore different options, which leads to more detailed answers and better analysis [YYZ+23]. As a result, TOT works well for NLP tasks that need step-by-step thinking and decisions.allows models to explore different possibilities, leading to more detailed responses and deeper analysis [YYZ+23].

2.3.3 In-Context-Learning

With In-Context Learning (ICL), LLMs can complete tasks without fine-tuning as they generate responses based on context given inside the prompt [BMR+20, RWC+19]. This context can be in the form of examples or descriptions. Usually, the number of examples is bound by its context window, which is the maximum number of "tokens" (words or pieces of words) including input and output that the model can process at once [Ber24]. A specific type of ICL is "one-shot-learning", where the prompt only gets one example [Li23, BMR+20].

ICL helps a model infer the correct response patterns and apply similar reasoning to new inputs by providing it with a few examples of "input-output pairs" right in the prompt [BMR+20]. When a prompt contains examples, the model is better able to adjust to tasks like summarization, translation, and classification. Rethinking role demonstrations makes sense [MLH+22].

If one wants to solve the question "What is the sum of the first 10 even numbers?"[Agg23] by using ICL, examples have to be added to before it to give the model more context. For instance, the prompt could look like this "Example 1: Question: What is the sum of the first 5 even numbers? How to solve it: The first 5 even numbers are: 2, 4, 6, 8, 10. Their sum is: 2 + 4 + 6 + 8 + 10 = 30. Answer: 30 Example 2: Question: What is the sum of the first 8 even numbers? How to solve it: The first 8 even numbers are: 2, 4, 6, 8, 10, 12, 14, 16. Their sum is: 2 + 4 + 6 + 8 + 10 + 12 + 14 + 16 = 72. Answer: 72 What is the sum of the first 10 even number ?"[Agg23]. The model would then proceed as in the examples. First, it lists the first ten even numbers, then it adds all of them up and finally, gets the answer 110.

Many NLP tasks benefit from using in-context learning (ICL), especially when quick learning is needed. For example, the model can give personalized answers or handle questions on a specific topic. It does this by finding patterns in the examples it sees and applying them to new inputs and because of this ICL is a helpful technique for prompt engineering [WZS+23, SIB+24].

2.3.4 Further enhancements

There have been recent developments in prompt engineering, which can improve prompts even further:

- **Decomposition:** Tasks are broken down into simpler sub-tasks using techniques such as Plan-and-Solve, which improves the reasoning of the model through structured thinking. An example for such a prompt, proposed by Schulhoff et al. [SIB+24], is: "Let's first understand the problem and devise a plan to solve it. Then, let's carry out the plan step by step."
- **Self-Criticism and Refinement:** Prompts in Self-Refine help the model to generate an answer, then critique it, and later improve it based on its own feedback in an iterative loop. For example, one could as the model after getting an answer back from the question "Is the above answer correct? If not, explain why and revise it."[SIB+24, Bha24].
- **Iterative Reasoning:** Techniques like Cumulative Reasoning allow the model to stop, evaluate intermediate steps, revise them if necessary, and explore alternative paths for more reliable results [ZYYY23, SIB+24].

2.3.5 Summary

Prompt Engineering strategies like Chain-of-Thought (COT), Tree-of-Thought (TOT), and In-Context Learning (ICL) are different methods that improve the performance of

LLMs (see Figure 2.5) [SIB+24]. Each strategy tackles different challenges in reasoning, which ensures that the approach fits the task for which it is used for.

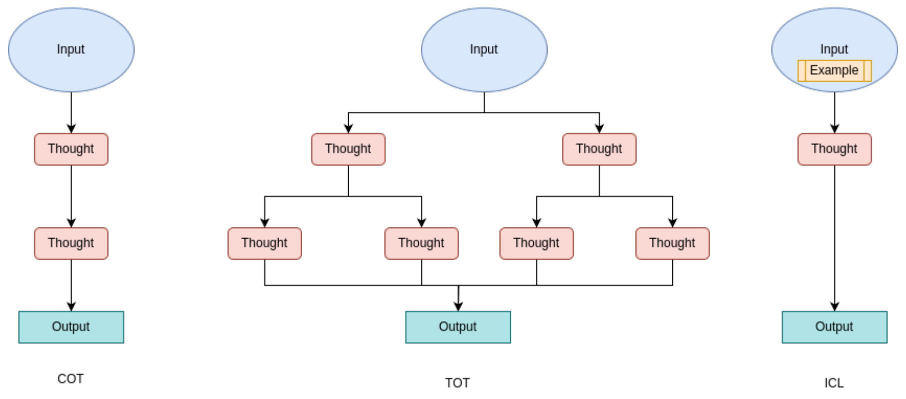


Figure 2.5: Prompting strategies

The **Chain-of-Thought (COT)** strategy helps LLMs to solve problems step-by-step by breaking them down into smaller parts. This does not only make it easier to follow the "thought" process of the model, but also leads to greater accuracy. Moreover, just adding prompts like *"Let's solve this step by step"* can help the model think more clearly and avoid rushing into wrong answers [KGR+22]. Usually, COT works very well for tasks that require logic, math, or everyday reasoning [WWS+22].

The **Tree-of-Thought (TOT)** strategy builds on this idea by letting the model explore different possible solutions in parallel, like branches growing from the same trunk. This is helpful for tasks that need creativity, planning, or flexible thinking, where there isn't just one right answer [YYZ+23]. The model explores different options, keeps the best ones, and ignores the rest [GOP+23]. Despite its computational demands, TOT enhances the breadth and depth of problem solving [SIB+24].

The **In-Context Learning (ICL)** strategy differs from both COT and TOT by using examples provided directly within the prompt. By giving input-output examples, ICL helps the model in applying its reasoning to tasks like classification, summarization, and domain-specific question answering [MLH+22]. But the key to its success is selecting relevant and high-quality examples.

Every strategy has its own advantages and disadvantages. COT tends to work well for tasks that follow a clear, step-by-step process like solving a math problem. On the other hand, TOT is better for situations that need more flexibility, such as brainstorming or exploring alternatives. ICL relies on examples, which makes it great for quickly adapting to new types of tasks. Knowing how these methods differ helps people pick the right one for their use case and make better use of what LLMs can do. They can be further improved by adding self-verification, iterative reasoning, or decomposition.

2.4 Related Work

Before Large Language Models (LLMs) were integrated, REST API testing had already seen major advancements.

Already in 2019,**RESTler** was introduced by Atlidakis et al. [AGP19], which uses stateful fuzzing and OpenAPI specifications for testing. In 2020, Viglianisi et al. [CZPC22] developed **RESTTESTGEN**, a tool that generates tests from API interface definitions by using Operation Dependency Graph (ODG). A year Later, Martin-Lopez et al. [MSR21] published **RESTest**, a model-based framework for creating thorough test cases.

Building on these approaches, Alonso et al. [AMS+23] introduced **ARTE**, which used database queries to create realistic test inputs to improve REST API testing. Arcuri et al. [Arc19b] developed **EvoMaster**, which combines black-box and white-box techniques with evolutionary algorithms to improve test coverage.

More recently, **RESTSpecIT** [DPP+24] used LLMs for black-box testing, which means that the internal structure of the API is unknown. It can understand how an API works without needing pre-written documentation. It automatically figures out API specifications and generates test data with minimal input. This reduces the effort needed to create detailed test cases, allowing APIs to be tested more quickly and thoroughly with little manual work. Both tools show how LLMs and AI can automate complex REST API testing. They improve test case generation, make testing more adaptable, and reduce manual effort, making API testing more flexible, efficient, and thorough.

In summary, REST API testing has improved from early tools like RESTler, RESTTEST-GEN, and RESTest to more advanced ones like ARTE and EvoMaster. These tools introduced new ways to test APIs like fuzzing, dependency-based test generation, and evolutionary algorithms. Newer tools like ARAT-RL and RESTSpecIT use LLMs to take API testing a step further. They use black-box testing and reinforcement learning to make testing more complete, flexible, and efficient. As LLMs improve, they could make testing easier, cut down on manual work, and increase test coverage.

2.4.1 Automated REST API Testing without LLMs

REST APIs were automatically tested in many different ways before to the raise of LLMs. They were thoroughly tested using different techniques in either white-box or black-box setting. Additionally, complex interactions were simulated with the help of methods such as model-based testing, evolutionary algorithms, and stateful fuzzing. Moreover, several test generation techniques were applied in order to produce realistic test scenarios. Sequence building, sampling, and constraint solving were among them (see Table 2.10).

Validation mechanisms frequently check status codes, detect server errors, and ensure schema compliance. Together, these strategies enable for thorough testing and identify issues that might be missed with simpler methods.

Tool	Methodology	Testing Type	Test Generation	Dependency/Mutation Handling	Metrics
RESTler (2019)	Stateful fuzzing	Black-box	Stateful request sequences	API endpoint dependencies	Status code checks (5xx), schema compliance
RESTest (2020)	Model-based testing	Black-box	Constraint-solving	Inter-parameter dependencies	Nominal + error test, Status code checks (5xx), schema compliance
EvoMaster (2021)	Evolutionary algorithms	Black-box + White-box	Sampling-based + mutation-based	Operation dependencies	Automated oracles (5xx), code coverage
ARTE (2022)	Semantic-based testing	Black-box	Uses knowledge bases to to create inputs for testing	Uses NLP for extracting keywords	Status code checks (4xx, 5xx), response validation
RestTestGen (2023)	graph-based testing	Black-box	Graph to define meaningful execution orders	Missing Required, Wrong Input Type, Constraint Violation	Status code checks (4xx), schema compliance, Nominal + error tests

Table 2.7: Comparison of Automated REST API Testing Tools

RESTler, the "first stateful REST API fuzzing tool", was created by Atlidakis et al. [AGP19] to evaluate the security and robustness of cloud services via their REST APIs. Unlike traditional fuzzers, RESTler focuses on stateful test generation, which means that it tries to discover service states that can only be accessed by a series of requests. This kind of testing mimics real-life scenarios. Since it is a black-box tool, it has no access to the internal code and uses the OpenAPI specification of the REST API to infer dependencies between endpoints. For example, if one request generates a resource ID needed by another, then RESTler ensures they are executed in the correct order. It then creates a Python-based test grammar, which uses the functions `restler_static` and `restler_fuzzable` to model request structures and introduce fuzzable inputs. This grammar helps RESTler build and check request sequences, starting simple and getting more complex, while checking responses (like HTTP 200) and using dependencies to extend tests. By adding dynamic data makes the tests even more varied. Moreover, RESTler uses smart search strategies like RandomWalk, BFS-Fast, and breadth-first search (BFS) to explore the API more effectively. These strategies allow RESTler to identify significant bugs, particularly server errors (HTTP 500). Testing in the real world has demonstrated that RESTler can identify security flaws [AGP19].

RESTest [MSR21, DPP$^+$24] is a black-box tool for generating tests using a "system model", which is an OpenAPI specification, and a "test model" that is a YAML file containing configurations. Using these two models, an abstract test case is generate, which are later instantiated into executable tests. A advantages feature of RESTest is supports inter-parameter dependencies, which are constraints between input parameters in an API operation that must be fulfilled to form valid requests. It translates these dependencies, which are defined in a constraint language, into an OpenAPI standard. It supports various test generation strategies, including adaptive random testing, constraint-based testing, and fuzzing. The latter uses constraint solvers (IDLReasoner) to ensure that the dependencies are respected. It focuses on dependencies between parameters.

RESTest uses random input generation and constraint solving to generate two kinds of tests: "faulty" tests, which violate the rules or omit necessary parameters, and "nominal" tests, which follow the rules to generate legitimate (2xx) replies. Moreover, RESTest uses "test oracles" to make sure the response does not return a server error (5xx), follows the correct format, and has the right status code depending on the type of test—whether it's a normal request or one meant to cause an error [MSR21].

EvoMaster [Arc19b] is a REST API testing tool with support for white-box as well as black-box testing which parse the OpenAPI specification as a starting point. In black box mode (EvoMasterBB) [KXSO22], generates random requests to test different API routes. To maximize code coverage in white-box mode (EvoMasterWB) [KXSO22], EvoMaster uses the evolutionary algorithm of MIO [Arc19a] and has access to the internal code. MIO evolves a population of test cases by selecting, mutating, and discarding tests based on their fitness for covering new targets. Additionally, EvoMasterWB uses two methods to generate test, which are sampling-based and mutation-based methods. Sampling-based tests are created from scratch with random or smart sampling, while mutation-based tests modify existing structures. Both modes use an automated oracle to detect service failures (5xx) [Arc19b, KXSO22].

ARTE (Automated REST Testing Engine) is a semantic-based testing tool to generate realistic input values for REST API testing [AMS+23]. It starts by analyzing the OpenAPI specification and uses external knowledge bases to retrieve predicates, which are are used for to create SPARQL queries. These queries are used to look for test inputs. In order to further improve the input quality, it uses regular expressions to filter and search for both valid and invalid values. ARTE has been integrated into RESTest and has helped increase the number of successful API calls during testing.

RESTTESTGEN automatically creates REST API test cases by modeling "producer-consumer" relationships using an Operation Dependency Graph (ODG) [CZPC22]. It starts with parsing the OpenAPI specification and extracts operations and schemas into internal models. Then the ODG is constructed, where the nodes represent the API operations, and edges show when the output of one operation is used as input for another. The graph captures how API operations depend on each other, which helps it to define meaningful execution order of the testing steps. Then, it generates test sequences, each made up of one or more test interactions. The input values for parameters are filled using various strategies like random generation, enum/example values, or previously seen values. For error testing, RESTTestGen applies mutation operators to introduce faults like missing required fields or invalid types. Finally, the responses are checked using test oracles: one evaluates HTTP status codes (2xx, 5xx), and another validates response schemas. Finally, results are saved using built-in writers—either as structured JSON reports or executable Java tests using REST-assured [CZPC22].

2.4.2 Automated REST API Testing with LLMs

The rise of LLMs has brought new tools and techniques that make REST API testing faster, more flexible, and more complete. Compared to traditional methods, these tools use LLMs to understand complex OpenAPI specifications, create detailed test cases, and fill in missing parts of the documentation. LLMs can handle complex situations and make testing more realistic, which helps to find hidden bugs. This section looks at key tools (see Table 2.8) that use LLMs for advanced API testing, such as rule extraction, guessing missing specs, and planning test tasks on the fly.

Tool	Methodology	Testing Type	Test Generation	Dependency/Mutation Handling	Evaluation Metrics
RESTGPT (Kim, 2023)	Rule extraction and specification enhancement	Black-box inference-based testing	Few-shot prompting with context-specific examples	Rule generator and specification enhancer	Rule extraction accuracy, precision, and coverage
RESTGPT (Luo, 2023)	Task decomposition for API interaction	Black-box dynamic online planning	Coarse-to-fine online planning	Planner, API selector, executor	Success Rate, Correct Path Rate
RESTSpecIT (2023)	OpenAPI specification inference and mutation	Black-box mutation-based testing	Focused mutation on routes and parameters	Masking, mutation process	Specification completeness, API behavior discovery

Table 2.8: Comparison of LLM-based REST API Testing Tools

The goal of **RESTGPT** et al. Kim [KSS+24] is to make REST API REST API testing better by extracting constraints from OpenAPI specifications. This ensures that APIs function correctly and adhere to intended behaviors. The study does not explicitly address security vulnerabilities, such as identifying potential exploits or weaknesses within the APIs. However, by improving the thoroughness of API testing, RESTGPT can indirectly contribute to identifying issues that could have security implications. It uses Prompt Engineering strategies, specifically few-shot prompting with context-specific examples, to help the LLM in extracting rules from API specifications and improves the ability of the model to produce precise and relevant test cases, which leads to better overall API testing quality [KSS+24].

The approach involves several stages (see Figure 2.6) [KSS+24]:

- **Rule Generator**: RESTGPT uses GPT-3.5 Turbo with special prompts to extract four types of rules, which are operational constraints, and parameter-related rules including constraints, types, formats, and examples. It uses few-shot prompting and context-specific examples to make the rules more accurate and useful.

- **Specification Enhancement**: RESTGPT augments OpenAPI specifications by merging machine-readable constraints with insights from natural language descriptions. This combined format provides enhanced specification documents that improve the coverage of API tests by including parameter constraints, types, formats, and examples.

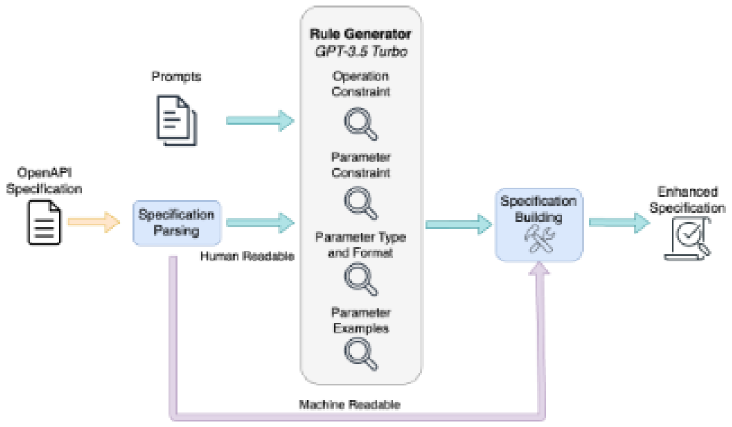


Figure 2.6: RESTGPT workflow et al. Kim [KSS+24]

When comparing RESTGPT to NLP2REST, RESTGPT shows better performance in rule extraction accuracy, precision, and coverage. The authors attribute these improvements to RESTGPT's use of context-aware language model prompts and structured rule extraction [KSS+24].

RestGPT by Song et al. [SXZ+23] enables LLMs to interact with real-world RESTful APIs. The main goal is to improve response parsing, task decomposition, and making complex REST API interactions easier. It achieves this by using coarse-to-fine online planning mechanism, which allows dynamic task planning and execution. Moreover, it is important to note that this paper does not focus on security testing. However, the authors recognize the importance of security and state that "security issues are also important in real-world applications" [SXZ+23].

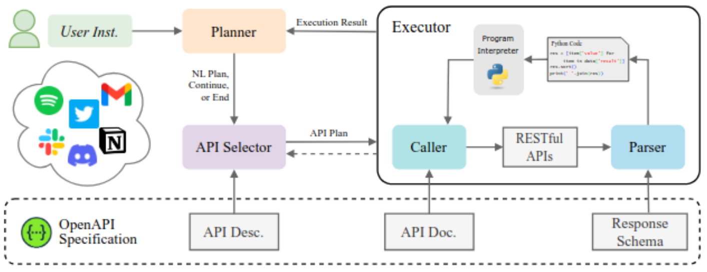


Figure 2.7: RestGPT workflow by Song et al. [SXZ+23]

RESTGPT consist of three modules, as shown in Figure 2.7):

- **Planner**: This module breaks down the user instructions into manageable subtasks. It can adjust to changes and uses a flexible "plan and execute" loop to refine tasks based on the user's goal.

- **API Selector**: The API Selector selects the right APIs to complete each task based on the sub-tasks that the Planner has created.
- **Executor**: This module handles responses and makes the actual API calls. It consists of a Caller that creates API requests and a Parser that uses the response to create Python code that reads and extracts data from the responses.

RestGPT can select APIs, break down tasks, and carry out plans step by step using a "coarse-to-fine" planning system. This allows it to adjust in real time based on feedback. For the evalution of effectiveness, the authors present the benchmark "RestBench", which includes two real-world scenarios (TMDB and Spotify). The reason why they are used is that they need multi-step API interactions. They measure performance using metrics like Success Rate and Correct Path Rate to see how well RestGPT handles complex tasks. The results show that RestGPT performs well, successfully managing complicated API interactions. Moreover, it is important to point out that RestGPT does not use a Prompt Engineering strategy but follows a structured approach to help the LLMs. The system includes three key components: Planner, API Selector, and Executor, which use carefully designed prompts to help the LLM break down tasks, select the right APIs, and interpret responses. By using context-aware prompts, this method effectively applies Prompt Engineering to achieve accurate and reliable results [SXZ+23].

Another framework called **RESTSpecIT** [DPP+24] uses LLMs (GPT-3.5) to test RESTful APIs and infer OpenAPI specifications. In particular, it focuses on finding server errors in RESTful APIs and undocumented routes and parameters. It only needs the API name and an LLM key to automatically create and edit HTTP requests with little user input (see Figure 2.8). While its main goal is to improve API testing and documentation, it can also accidentally uncover security issues, like misconfigured or hidden endpoints. It is important to note that RESTSpecIT uses Prompt Engineering strategies to assist LLMs in generating HTTP requests. For this purpose, it uses "in-context prompt masking technique". This approach enables the tool to infer API specifications effectively without requiring fine-tuning of the model. However, it is important to mention that this framework is not made for security testing.

RESTSpecIT is made up of four essential phases:

- **User Initialization and Input**: For initialization, it needs an LLM key, an API name and a small configuration file. Then it generates an initial OpenAPI specification and requests that the LLM gather the required API data (such as the description, URLs, and terms of service).
- **Masking and Mutation Process**: It selects HTTP request seeds and applies mutation operators like adding parameters or changing routes to mask specific parts of the request. Then, it uses the LLM to generate new versions of those parts.

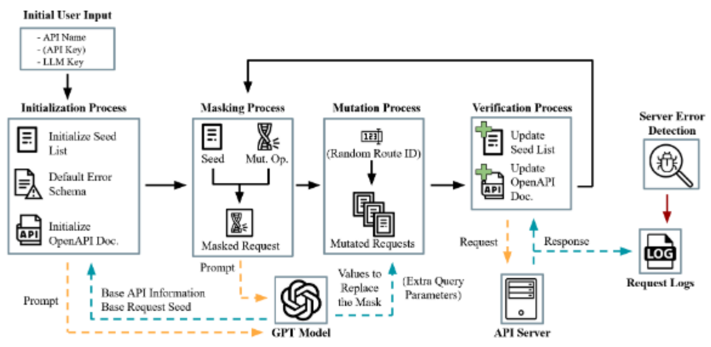


Figure 2.8: RESTSpecIT workflow et al. [DPP+24]

The modified requests are checked to make sure they meet certain conditions, like getting a 2xx status code and not containing error messages.

- **Specification Inference**: It updates the OpenAPI specification based on the validity of the request. If the request was valid, it is added to the seed list and the route is added to the specification. If new parameters for an endpoint are found, they are added to the entry of the route in the documentation. Concrete values are recorded in form of examples in the OpenAPI specification.

- **Mutation Strategies**: The tool uses three techniques (Focus Routes, Focus Parameters, and Focus All) to analyze the API structure more thoroughly. Each method applies specific mutation changes step by step to discover new API behaviors.

The evaluation of RESTSpecIT shows its capability to infer API structures and generate comprehensive OpenAPI specifications with minimal input, demonstrating its effectiveness in testing and documenting RESTful APIs through LLM-powered mutation and validation.

PentestGPT was developed by Deng et al. [DLMV+24] and uses LLMs for automated penetration testing. It tackles important issues with context maintenance in complex testing situations [ISCK24].

PentestGPT consists of three essential modules, where each serves a specific purpose [DLMV+24]:

1. **Reasoning Module**: It refreshes the Pentesting Task Tree (PTT), which lays out testing methodologies, and keeps a high-level perspective.

2. **Generation Module**: It converts the Reasoning Module's strategic plans into test commands.

3. **Parsing Module**: It processes and simplifies wordy outputs from various sources to help with data interpretation and decision making.

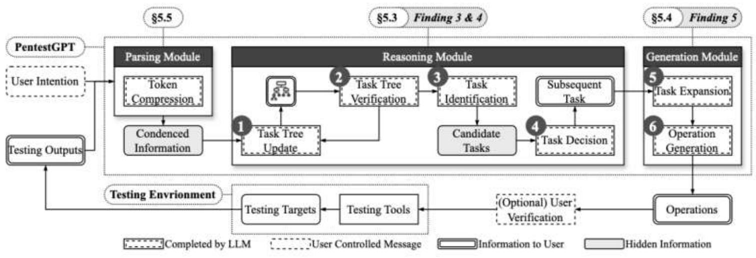


Figure 2.9: PentestGPT workflow et al. Deng al. [DLMV+24]

PentestGPT operates through several phases (see Figure 2.9) [DLMV+24]:

- **Initialization and Configuration**: In this phase, the initial test parameters and target definitions are set up.
- **Task Execution and Monitoring**: During this phase, tasks are controlled and carried out in real time according to PTT updates and data.
- **Results Analysis and Feedback Loop**: This stage involves constant analysis and improvement of the outputs to guarantee that the PTT accurately reflects the testing state.

In automated penetration testing, it has performed better than traditional LLM applications. It has been extensively tested in real-world settings, including established benchmarking platforms and HackTheBox[12] challenges, where it has demonstrated significant gains in task completion rates and overall efficacy [ISCK24].

It shows that LLMs can offer improvements in cybersecurity by making penetration testing processes more precise and efficient. Its release as an open source tool enables continuous development and encourages community contributions, which are vital for advancing cybersecurity testing methodologies.

2.4.3 Comparison to this thesis

In comparison to the related work, this thesis explores how Prompt Engineering strategies can be utilized to enhance LLM performance in REST API testing and looks at their potential to dynamically detect security vulnerabilities. While existing automated testing tools, whether LLM-based or not, focus primarily on robustness and functional testing, they largely overlook targeted security vulnerability detection. Although PentestGPT is oriented towards penetration testing, it separates the documentation process from the testing itself. This gap forms the basis of our work.

[12]`https://www.hackthebox.com`

Point	Existing Tools	This Thesis
Focus Area	Functional testing and robustness.	Detecting security vulnerabilities in REST APIs.
Specification Generation	Generates OpenAPI specifications using tools like RESTSpecIT.	Generates OpenAPI specifications focused on security risks using the "WebAPIDocumentation" approach.
Testing Objectives	Prioritizes functional coverage and specification completeness.	Targets OWASP Top 10 risks, such as injection attacks and broken authentication.
Methodology	Uses standard strategies for robustness and compliance.	Integrates LLMs with Prompt Engineering for dynamic security test cases.
Advancements	Limited in addressing security vulnerabilities.	Bridges functional testing and security assessment for comprehensive testing.

Table 2.9: Comparison Between Existing Tools and This Thesis

Building upon concepts from tools like RESTSpecIT, we similarly generate OpenAPI specifications dynamically by querying the REST API using the "WebAPIDocumentation" approach (see Section 3.2.1). However, our approach's objectives and focus are completely different. Compared to other tools that focus on functional coverage or completeness of specifications, our approach is specifically designed to identify and evaluate security flaws.

Our testing framework focuses on the OWASP Top 10 security risks (see Table 3.1), which include Broken Authentication, Injection Attacks, and Security Misconfigurations. We combine specialized Prompt Engineering techniques with LLMs to enable the dynamic generation of targeted test cases to actively search for these vulnerabilities. This approach provides a comprehensive framework for safeguarding REST APIs and expands the range of currently available tools by bridging the gap between functional testing and security assessment.

2.4.4 Summary

Traditional REST API testing automation tools like RESTler and EvoMaster depend on structured approaches such as stateful fuzzing, evolutionary algorithms, and graph-based analysis. These approaches depend heavily on predefined rules, API specifications (OpenAPI), and static dependency management. In contrast, LLM-using tools, such as RESTGPT and RESTSpecIT, dynamically interpret API documentation, generate test cases, and even infer missing specifications. This ability introduces context-aware reasoning and adaptive test generation, which were previously absent in rule-based systems.

Test generation is a key area where LLM-based tools offer significant advancements. They use dynamic techniques such as few-shot prompting (RESTGPT by Kim), task decomposition (RestGPT by Luo) and mutation strategies (RESTSpecIT). These approaches enable LLMs to create customized test cases based on learned API behaviors. LLMs also make it possible to improve pentesting (PentestGPT). In comparison, traditional tools that do not use LLMs rely on static methods like building sequences, sampling, solving

Point	Without LLMs	With LLMs
Methodology and Approach	Uses fixed rules and tools like fuzzing or evolutionary algorithms.	Understands docs and fills gaps using language models.
Testing Type	Uses black-box and white-box testing with stateful flows.	Uses adaptive black-box testing based on context.
Test Generation	Builds tests using static sequences and constraints.	Uses prompts and mutations to create tests on the fly.
Dependency and Mutation Handling	Uses static rules to handle dependencies and mutations.	Learns and adjusts dependencies dynamically.
Evaluation Metrics	Measures code coverage, schema checks, and status codes.	Adds metrics like rule matching and behavior detection.
Strengths	Works well with full API specs and stable systems.	Flexible for incomplete docs and complex APIs.
Limitations	Needs full specs and struggles with changing APIs.	Depends on LLM quality and needs more compute.
Use Cases	Best for stable, well-documented APIs.	Suited for exploring and testing evolving APIs.

Table 2.10: Comparison of API Testing Tools With and Without LLMs

constraints, and applying mutations. These methods are not flexible and cannot easily adjust to complex or changing situations.

Although API testing without LLMs has proven to be highly reliable in structured environments with well-defined specifications, it is most effective for detecting common bugs and ensuring compliance. These tools use clear metrics like code coverage and schema validation that makes them reliable in predictable situations. However, LLM-based tools, although relatively new and less extensively validated, show greater adaptability. They work very well for testing APIs with incomplete or poorly documented specifications. This is because they can infer missing information, generate test cases dynamically, and explore complex edge cases.

Both approaches have drawbacks. Testing tools without LLMs depend on complete and accurate API specifications. This makes them not work well when faced with undocumented or dynamically evolving APIs. Their testing coverage is often constrained by predefined methodologies and static rules.

On the other hand, LLM-based tools heavily depend on how well the language model understands the prompt, which can lead to mistakes if the context is not clear. Moreover, they need a lot of computing power, which makes them more expensive to run and they need a lot resources.

In short, traditional tools are strong in well-defined settings and follow clear rules, while LLM tools offer more adaptability and can handle harder testing problems. Used together, they can provide more complete REST API testing (see Table 2.10).

CHAPTER 3

Approach

This elaborates on the technical/ formal contributions, conceptual approach, and implementation of this thesis.

3.1 Technical/Formal Contribution

This thesis aims to evaluate the effectiveness of Large Language Models (LLMs) in generating test cases for REST APIs, identify the most effective Prompt Engineering techniques for this purpose, and assess their ability to detect potential failure points or vulnerabilities with accuracy. In order to answer these questions, an extensive literature review was conducted, and preexisting work like RESTGPT and RESTSpecIT were closely examined and tested before implementing our use case in HackingBuddyGPT[1]. This was done to find out the drawbacks of the currently existing solutions. The following conclusions could be drawn from these two tools:

- The output of an LLM is dependent on the prompt's structure, specificity, and clarity because well-written prompts direct the model to provide precise and pertinent answers.
- Good REST API testing depends on a good OpenAPI specification.
- None of them experimented with Prompt Engineering and explored how this could improve the performance of the models.
- The tools for automatic test generation did not specifically test if the REST APIs had security vulnerabilities.

[1]`https://github.com/ipa-lab/hackingBuddyGPT`

Therefore, in this work, we first focused on automating the way of creating a good OpenAPI specification with the help of the LLM similar to how it was done in RESTSpecIT in a black-box setting. This OpenAPI specification was generated by continuously giving feedback to the model. Then this OpenAPI specification was used in order to properly test the REST API for this we used the OWASP Top 10 security risks (See Table 3.1).

Vulnerability	Description	Measurement Methods
Authentication Issues	Testing credential-based authentication, token validity, and privilege escalation	Query endpoints for login, verify token/session validity, simulate brute-force attacks
Authorization Issues	Unauthorized access or privilege escalation vulnerabilities	Simulate role-based access control (RBAC) violations, test restricted endpoint access
Injection Attacks	SQL, command, or NoSQL injection vulnerabilities	Dynamically test inputs for unsanitized parameters, automate SQL/command injection scenarios
Input Validation	Testing for improper input handling	Test for invalid formats, boundary values, and parameter injection
Error Handling & Information Leakage	Leaking sensitive error messages or data	Trigger errors intentionally, analyze responses for sensitive information
Sensitive Data Exposure	Unencrypted data transmission or misconfigured security headers	Inspect API responses and headers for exposed sensitive data or missing encryption
Rate Limiting & DoS Protection	Protection against brute-force or DoS attacks	Simulate high request loads, check for rate-limiting mechanisms
Special Authentication Mechanisms	Testing advanced authentication setups (e.g., multi-factor, token-based)	Evaluate multi-factor authentication flows, tamper with token structures
Endpoint Categorization	Analyzing categorized endpoints for structural issues	Use OpenAPI parsing to classify and test endpoints systematically

Table 3.1: API Security Vulnerabilities and Measurement Methods Implemented in Code

3.2 Conceptual Approach

In this thesis, the automation of the generation of OpenAPI specifications and the execution of security tests was achieved through two use cases: WebAPIDocumentation and WebAPITesting. First, the OpenAPI specifications were created using WebAPIDocumentation. Then WebAPITesting is used to executes tests based on the specification.

3.2.1 WebAPIDocumentation

WebAPIDocumentation was used to create thorough and accurate OpenAPI specifications, addressing common issues such as missing information and inconsistencies in the documentation. The documentation process was divided into two main stages:

1. **Explore:** In this phase, the model sends GET requests to random endpoint paths based on the current step. Only GET requests are used to ensure that the state of the REST API remains unchanged. GET methods are HTTP safe methods because they only retrieve information and do not change anything. The endpoints that return 404 (not found) or 400 (bad request) indicate invalid paths or malformed requests. These enpoints are recorded so that they can be explored in the Exploit phase. The model systematically queries endpoints to explore the API's structure without modifying or deleting data. The endpoints are systematically queried:

   a) Find root-level resource endpoints

b) Find instance-level endpoints

c) Find sub-resource endpoints

d) Find endpoints with query parameters

e) Find related resource endpoints

f) Find multi-level-endpoints

2. **Exploit:** In this phase, the model should already have discovered all accessible endpoints based on the responses to the requests. The endpoints that were discovered in the previous phase that returned a 400 or 404 error are now tested with alternative HTTP methods like POST, DELETE, PUT, and PATCH. Moreover, all the already found endpoints are tested. Thus, any methods not previously tested are now explored to determine their compatibility with the identified endpoints.

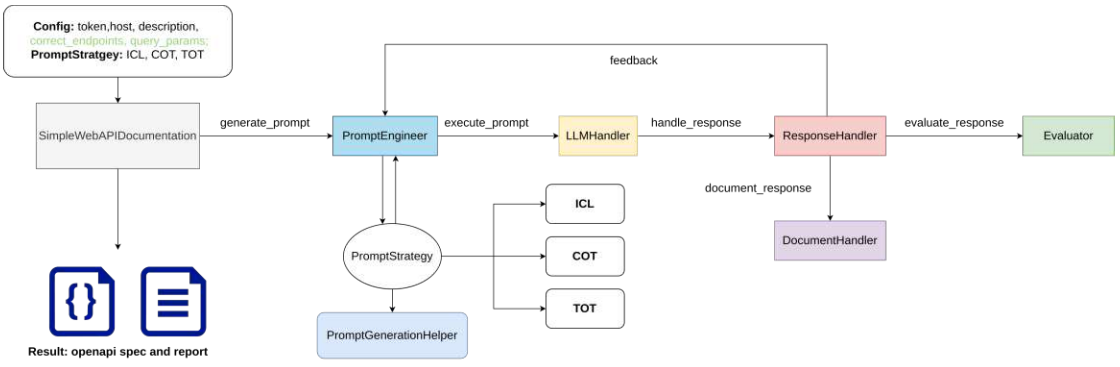


Figure 3.1: SimpleWebAPIDocumentation

The workflow of this looks like the following (see Figure 3.1):

1. A configuration with minimal information (description, url, and token if needed) and the type of prompt strategy is passed to the program

2. The `PromptEngineer` utilizes the `PromptGenerationHelper` of the chosen `PromptStrategy` to generate an adequate prompt based on the phase it is in.

3. The prompt is passed to `LLMHandler` and executed.

4. The `ResponseHandler` processes the model's response. If the response indicates a new action or a new path, the corresponding HTTP request is made and assessed. Successful requests (status code 200 without error messages) are logged as correct in the prompt history. If a 400 error occurs, the error message is recorded to guide improvements. Unsuccessful actions are noted in the prompt history. Repeated failures or unsuccessful paths trigger hints or suggestions for common endpoints to be tested.

5. The result of the executed HTTP request is given to the `DocumentHandler`, which documents the findings in an OpenAPI specification for all successfully found endpoints. In case a method for an endpoint is invalid, but the endpoint is correct, this is also saved so that the error behavior is known for the testing procedure.

6. In order to evaluate the program and compare its performance to already existing tools, the `Evaluator` calculates and saves the routes found via GET, the parameters found, and the false positive query parameters based on the correct_endpoints and query_params provided in the configuration.

7. At the end of the run, the finished OpenAPI specification and a report about the effectiveness of the LLM are the result.

3.2.2 WebAPITesting

SimpleWebAPITesting checks both the functionality and security of the REST API. This method tests how well the API works and how secure it is against major threats, focusing especially on the OWASP Top 10 security risks, a list of the most critical web application security threats (see Table 3.1). By addressing these risks, we aim to ensure that the API is secure and reliable for practical use.

The workflow of this looks like the following (see Figure 3.2):

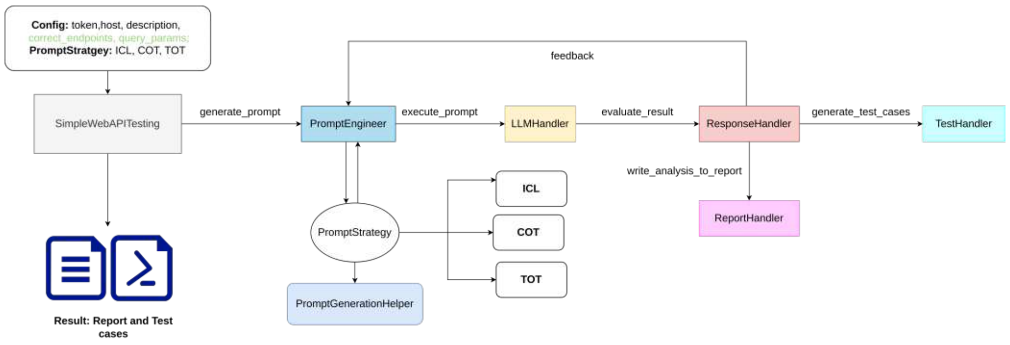


Figure 3.2: SimpleWebAPITesting

1. A configuration with minimal information (description, url, and token if needed) and the type of prompt strategy is passed to the program

2. The `PromptEngineer`, which is responsible for managing the prompt generation process, uses the `PromptGenerationHelper`, which offers helping methods for the prompt generation such as parsing, of the chosen `PromptStrategy` (Cot, Tot, Icl; See Section 3.2.3) to generate an adequate prompt based on the phase it is in.

3. The prompt is passed to the `LLMHandler` , which handles all operations related to the LLM like execution and checking the prompt.

4. The `ResponseHandler`, which is responsible for parsing the response, processes the result of the model response. The result is analyzed by the `ResponseAnalyzer`, which checks the status code, body and other information of the HTTP request.

5. The analysis from the `ResponseAnalyzer` is written by the `ReportHandler` to a report file.

6. The `TestHandler` generates test cases and writes them into a test file that can be executed later.

3.2.3 Prompt Generation Process

The prompts are generated through Prompt Engineer, which creates the prompts with the help of PromptGenerationHelper based on PromptStrategy (see Figure 3.3). The PromptStrategy can be one of three types: Tree-of-Thought (TOT), Chain-of-Thought (COT), and In-context Learning (ICL)(see Section 12).

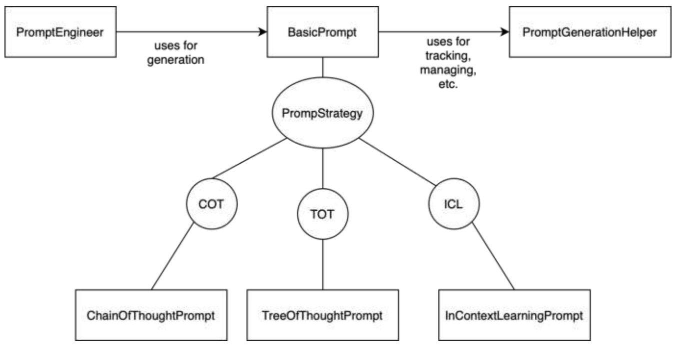


Figure 3.3: Prompt Generation process

- The **Prompt Engineer** creates a new prompt using the current interaction context and strategy with the help of the **Prompt Class** . The generated prompt is added to the history and the updated list is returned.
- The **Prompt Class** generated prompts based on the type of prompt class and the current interaction context. The Prompt Generation Helper is used to make prompt generation easier.
- The **Prompt Generation Helper** offers helper functions that make prompt generation easier. It tracks the endpoints, generates test prompts, and helps spot missing parts like endpoints or methods. Based on the results, it adjusts the testing process to make sure the entire API is tested and well-documented.

There are two different contexts for the prompt: Documentation or Pentesting. For both contexts, there are base prompts that are modified to fit the prompt strategy.

Chain-of-Thought Prompts

Chain-of-Thought prompts break tasks into steps to improve clarity and reasoning. These prompts break down tasks into smaller steps, which helps the model solve problems more accurately [WWS+22]. Moreover, they include conditional steps, so that the model can adjust its approach like retrying or improving a response if needed [R24]. When needed dynamic branching is used, which lets the model try different paths and check its progress before continuing, which leads to more flexible and effective problem-solving [SIB+24] (see Figure 3.4b).

Tree-of-Thought Prompts

Tree-of-Thought (ToT) prompts help the model think through problems step by step, using a branching structure with checkpoints along the way [Hul23, SIB+24]. It starts with a root, then explores different reasoning paths, where each branch represents a decision or step. If the model gets unexpected results, it can adjust using conditional logic [SIB+24]. ToT also uses self-checks to spot and fix mistakes as it goes [WZX+23]. The outputs are clearly structured to make them readable and useful, helping models to handle complex tasks effectively (see Figure 3.4c).

In-Context Learning Prompts

In-Context Learning (ICL) prompts guide tasks by using past examples. The model retrieves steps and outcomes from previous tasks and adapts them to fit the current context. For example, it might include details like API properties and response examples to improve accuracy. The prompts are dynamically adjusted to reference past examples and ensure clarity, using structured formats like JSON responses to make them more useful. This approach helps the model learn from past interactions and solve new tasks more effectively (see Figure 3.4a).

Phase: Setup tests
Create an account by sending an appropriate HTTP request to the correct endpoint: `/identity/api/auth/signup` with these credentials of user:
`{"name": "anna murphy", "email": "amy69@example.net", "number": "6153733498", "password": "2SQAhdDvR1"}.`
Request body should be in `application/json` format and look similar to this:
`{"name": "{{name}}", "email": "{{email}}", "number": "{{phone}}", "password": "{{password}}"}.`
Example:
`{"email": {"type": "string", "example": "Cristobal.Weissnat@example.com"}, "name": {"type": "string", "example": "Cristobal.Weissnat"}, "number": {"type": "string", "example": "6915656974"}, "password": {"type": "string", "example": "5hmb0gvyC__hVQg"}}.`

(a) Example of In-Context Learning

Phase: Setup tests
`Let's think step by step.`
Steps:
Step 1:
Create an account by sending an appropriate HTTP request to the correct endpoint: `/identity/api/auth/signup` with these credentials:
`{"name": "mary carpenter", "email": "samuel47@example.com", "number": "7394089726", "password": "gZCTNNqj4e"}.`
Request body should be in `application/json` format and look like:
`{"name": "{{name}}", "email": "{{email}}", "number": "{{phone}}", "password": "{{password}}"}.`

(b) Example of Chain-of-Thought

Root Objective: Setup tests
Tree of Thought:
Branch 1:
Step: "Imagine three different experts are answering this question.
All experts will write down one step of their thinking, then share it with the group.
Then all experts will proceed to the next step.
If any expert realizes they're wrong at any point, they will leave.
The question is:" [Hul23, LZ25]
Create an account by sending an appropriate HTTP request to the correct endpoint: `/identity/api/auth/signup`.
Use the following credentials:
`{"name": "denise lin", "email": "tina89@example.org", "number": "2188816795", "password": "7l7T7g20Ts"}.`
Request Body: The request should be in `application/json` format and look like:
`{"name": "{{name}}", "email": "{{email}}", "number": "{{phone}}", "password": "{{password}}"}.`
Security: Ensure that the correct expected code is received, confirming that access is restricted and no sensitive data is included in the response body.
Expected Response Code: `200 OK`.

(c) Example of Tree-of-Thought, ToT prompting taken from [Hul23]

Figure 3.4: Examples of prompts for REST API pentesting

3.3 Implementation

The implementation of this thesis was done within the "web_api_testing" use case of the publicly available open-source project **HackingBuddyGPT** [2] [HC23]. It is a GPT-powered tool designed for ethical hacking and API security testing and automates tasks such as vulnerability assessment, penetration testing, and test case generation, with a strong emphasis on addressing the OWASP Top 10 risks. LLMs simplify security testing workflows and increase efficiency.

[2] `https://github.com/ipa-lab/hackingBuddyGPT`

3.3.1 Prompt Generation

The prompt generation process consists of three components `PromptEngineer`, `Prompt`, and `PromptGenerationHelper` (see Figure 3.5).

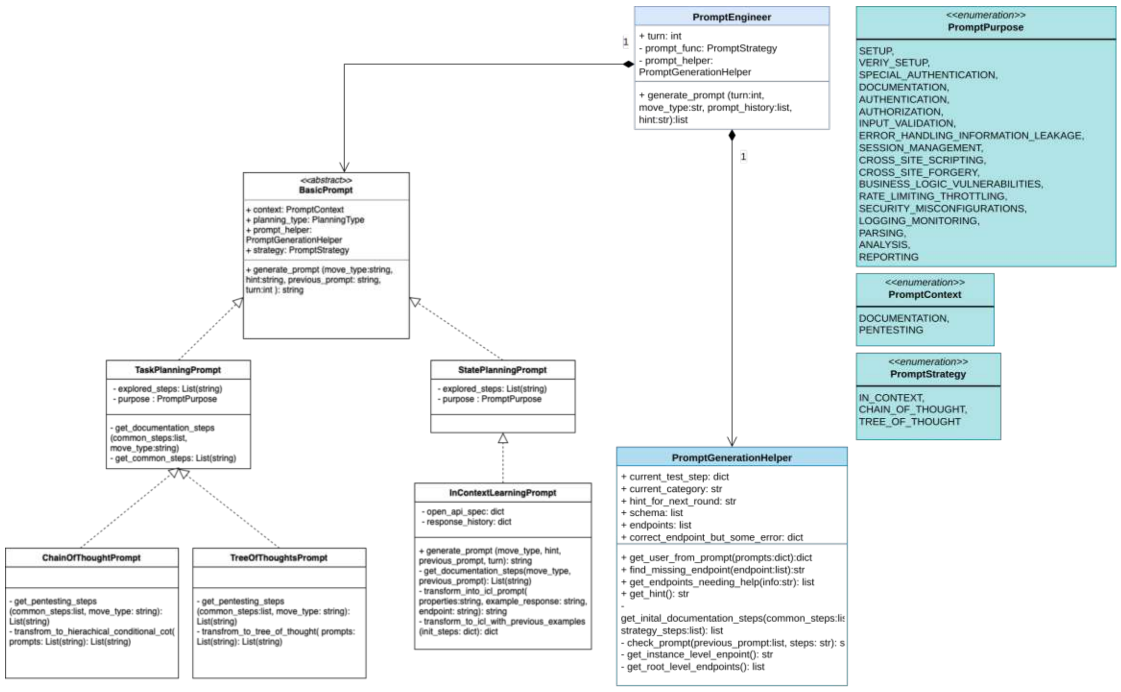

PromptEngineer
+ turn: int
- prompt_func: PromptStrategy
- prompt_helper: PromptGenerationHelper
+ generate_prompt (turn:int, move_type:str, prompt_history:list, hint:str):list
1
1
<<enumeration>> PromptPurpose
SETUP, VERIY_SETUP, SPECIAL_AUTHENTICATION, DOCUMENTATION, AUTHENTICATION, AUTHORIZATION, INPUT_VALIDATION, ERROR_HANDLING_INFORMATION_LEAKAGE, SESSION_MANAGEMENT, CROSS_SITE_SCRIPTING, CROSS_SITE_FORGERY, BUSINESS_LOGIC_VULNERABILITIES, RATE_LIMITING_THROTTLING, SECURITY_MISCONFIGURATIONS, LOGGING_MONITORING, PARSING, ANALYSIS, REPORTING
<<enumeration>> PromptContext
DOCUMENTATION, PENTESTING
<<enumeration>> PromptStrategy
IN_CONTEXT, CHAIN_OF_THOUGHT, TREE_OF_THOUGHT
<<abstract>> BasicPrompt
+ context: PromptContext
+ planning_type: PlanningType
+ prompt_helper: PromptGenerationHelper
+ strategy: PromptStrategy
+ generate_prompt (move_type:string, hint:string, previous_prompt: string, turn:int): string
TaskPlanningPrompt
- explored_steps: List(string)
- purpose : PromptPurpose
- get_documentation_steps (common_steps:list, move_type:string)
- get_common_steps: List(string)
StatePlanningPrompt
- explored_steps: List(string)
- purpose : PromptPurpose
ChainOfThoughtPrompt
- get_pentesting_steps (common_steps:list, move_type: string): List(string)
- transfrom_to_hierachical_conditional_cot(prompts: List(string): List(string)
TreeOfThoughtsPrompt
- get_pentesting_steps (common_steps:list, move_type: string): List(string)
- transfrom_to_tree_of_thought(prompts: List(string): List(string)
InContextLearningPrompt
- open_api_spec: dict
- response_history: dict
+ generate_prompt (move_type, hint, previous_prompt, turn): string
- get_documentation_steps(move_type, previous_prompt): List(string)
- transform_into_icl_prompt(properties:string, example_response: string, endpoint: string): string
- transform_to_icl_with_previous_examples (init_steps: dict): dict
PromptGenerationHelper
+ current_test_step: dict
+ current_category: str
+ hint_for_next_round: str
+ schema: list
+ endpoints: list
+ correct_endpoint_but_some_error: dict
+ get_user_from_prompt(prompts:dict):dict
+ find_missing_endpoint(endpoint:list):str
+ get_endpoints_needing_help(info:str): list
+ get_hint(): str
- get_inital_documentation_steps(common_steps:lis strategy_steps:list): list
- check_prompt(previous_prompt:list, steps: str): s
- get_instance_level_enpoint(): str
- get_root_level_endpoints(): list

Figure 3.5: Prompt Generation Process

The `PromptEngineer` class serves as the core component for creating and managing prompts in the **HackingBuddyGPT** project. It is responsible for generating adaptive and context-aware prompts to make tasks such as API testing and vulnerability assessments easier. The workflow looks like the following (see Algorithm 3.1):

- **Initialization:** The `PromptEngineer` is initialized with a specific prompt strategy, prompt context, REST API information, and an empty OpenAPI specification.The strategies and contexts that the component can use are predefined and represented as enums. For instance, the context can be `DOCUMENTATION` or `PENTESTING`, and the strategy can be set to `CHAIN_OF_THOUGHT`, `TREE_OF_THOUGHT`, or `IN_CONTEXT_LEARNING`.
- **Strategy Handling:** The prompt engineer creates prompts using the appropriate prompt class based on the selected approach (for more details, see subsection 12).
- **Prompt Generation:** The `generate_prompt` method creates a new prompt based the current prompt context, purpose and strategy. It updates the purpose of the prompt, which is important for the WebAPITesting usecase as it specifies the

goal of the test (e.g., checking security or functionality). The method then adds the new prompt to the history, keeps track of the previous prompt, increases the interaction count to reflect progress in the testing process, and finally returns the updated history of prompts.

Algorithm 3.1: Generate Prompt with Context and Strategy

```
Input: turn: Interaction number, move_type: Interaction type ("explore"), prompt_history:
       Optional, hint: Optional ("")
Output: Updated prompt_history with the new prompt and response
1 Generate prompt using the generate_prompt(move_type, hint, self._prompt_history,
    turn=0) from the correct Prompt class
2 Update purpose of the prompt
3 Append {"role": "system", "content": prompt} to prompt_history
4 Update self.previous_prompt and increment self.turn
5 return prompt_history
```

Algorithm 3.2: Generate In-Context Learning (ICL) Prompt

```
Input: properties: Dictionary of object properties
example_response: Example API response in JSON format
endpoint: Target endpoint of the API
Output: prompt: Generated prompt string
1  Initialize: Start with API core information as prompt
2      Append "REST API: example_response.keys() endpoint"
3      Append "This API retrieves objects with the following properties:"
4      foreach prop, details in properties.items() do
5      |   Append "- prop: details['type'] (e.g., details['example'])" to prompt
6      end
7      Append "Example Response:" to prompt
8      if example_response is not empty then
9      |   Extract example_key as the first key in example_response
10     |   Format the example_response[example_key] as JSON and append to prompt
11     end
12     return prompt
```

For multi-step tasks Algorithm 3.6 This creates efficient context-sensitive prompts, which are essential for automating tasks such as endpoint discovery, and security assessments in web API testing.

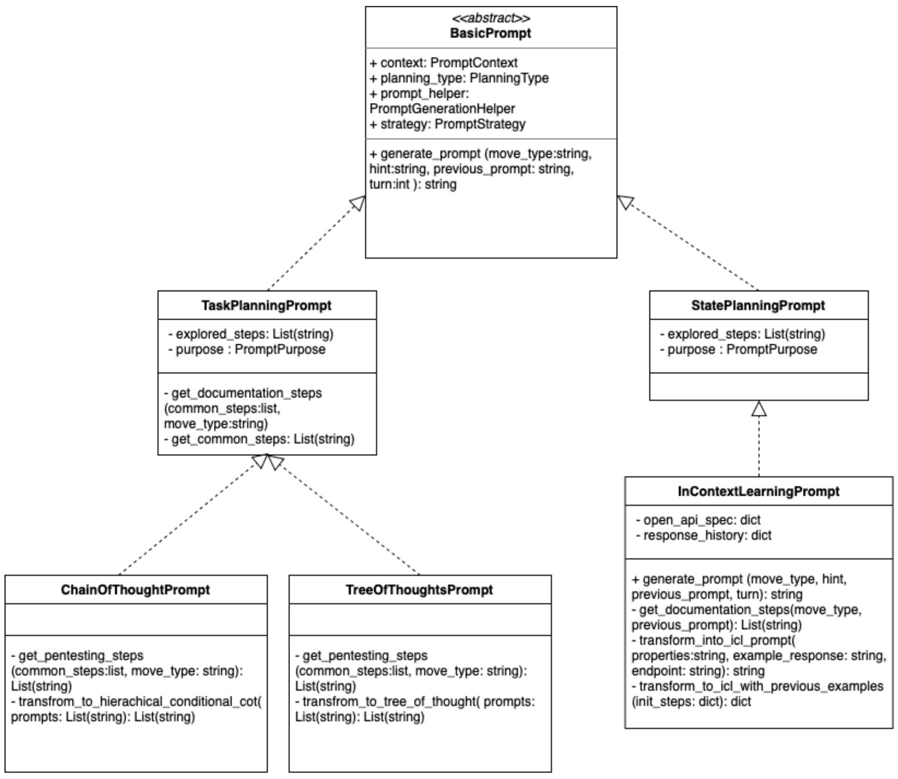
The `PromptGenerationHelper` class is essential for API testing, handling the management and analysis of endpoint data, generating testing prompts, and tracking the testing process. It efficiently keeps track of endpoints that have been discovered, tested, or failed, along with their associated HTTP methods. The class also designs prompts tailored to meet the specific needs of the API schemas and the progress of ongoing tests. It helps identify and fill gaps by spotting missing endpoints, HTTP methods, or query parameters and offers customized hints for these missing elements.

Additionally, the class dynamically adjusts the testing approach based on the outcomes of previous tests. It provides recommendations for future steps, while keeping the testing strategy agile and informed. Moreover, makes sure that prompts are within set limits and are clear and complete. Overall, the `PromptGenerationHelper` class helps automate and improve web API testing by making it easier to explore, analyze, and document how the API works.

The `Prompt` class is responsible for the actual generation of the prompt and will be described in the following subsection.

Prompts

The prompts are all derived from the `BasicPrompt`, which is the abstract class that encodes all the important attributes and functions of a prompt (see Figure 3.6). `TaskPlanningPrompt` and `StatePlanningPrompt` are subclasses of `BasicPrompt`



<<abstract>>
BasicPrompt
+ context: PromptContext
+ planning_type: PlanningType
+ prompt_helper: PromptGenerationHelper
+ strategy: PromptStrategy
+ generate_prompt (move_type:string, hint:string, previous_prompt: string, turn:int): string
TaskPlanningPrompt
- explored_steps: List(string)
- purpose : PromptPurpose
- get_documentation_steps (common_steps:list, move_type:string)
- get_common_steps: List(string)
StatePlanningPrompt
- explored_steps: List(string)
- purpose : PromptPurpose
InContextLearningPrompt
- open_api_spec: dict
- response_history: dict
+ generate_prompt (move_type, hint, previous_prompt, turn): string
- get_documentation_steps(move_type, previous_prompt): List(string)
- transform_into_icl_prompt(properties:string, example_response: string, endpoint: string): string
- transform_to_icl_with_previous_examples (init_steps: dict): dict
ChainOfThoughtPrompt
- get_pentesting_steps (common_steps:list, move_type: string): List(string)
- transfrom_to_hierachical_conditional_cot(prompts: List(string)): List(string)
TreeOfThoughtsPrompt
- get_pentesting_steps (common_steps:list, move_type: string): List(string)
- transfrom_to_tree_of_thought(prompts: List(string)): List(string)

Figure 3.6: UML diagram prompts

and describe different types of prompts. The `TaskPlanningPrompt` focuses on achieving specific tasks or goals, which typically involves sequences of actions or steps that need to be executed to complete a task. In contrast, the `StatePlanningPrompt` focuses more on how the system moves between states, using the conditions and effects of actions to guide that process.

The `ChainOfThoughtPrompt` and `TreeOfThoughtPrompt` are subclasses of the `TaskPlanningPrompt` as they work by breaking the task down into several steps.

Chain-of-Thought prompts are designed to support linear, sequential reasoning where each step builds on the previous one, whereas Tree-of-thought prompts are designed to create prompts that help explore different possibilities or solutions for a given task by using branches. Each branch stands for a different solution to the problem.

The `InContextLearningPrompt` is a subclass of the `StatePlanningPrompt`. These types of prompts are designed to learn and adapt according to the specific situation it encounters. This is achieved by adding the examples directly into the prompt. In doing so, the prompt gathers context-specific information, which improves its ability to provide appropriate and tailored responses. This method uses past examples as a guide, allowing the system to better understand and respond to similar situations in the future. Essentially, it is like teaching the prompt through direct experience, which makes it more effective in handling comparable scenarios.

Chain-of-Thought

The `ChainOfThoughtPrompt` class implements the Chain-of-Thought (COT) strategy, extending on the `TaskPlanningPrompt` class as it breaks the task down into linear steps. The prompts are customized to specific prompt contexts, such as `DOCUMENTATION` and `PENTESTING`. The steps are enumerated, the completed steps are tracked, and if unexplored scenarios occur, the prompt is adjusted.

Algorithm 3.3: Transform Test Case to Hierarchical-Conditional Structure

```
   Input: test_case: Dictionary with fields like 'objective', 'steps', 'security'
   purpose: Purpose of the test case
   Output: transformed_case: A structured test case with phases and conditions
1  Initialize: transformed_case with "phase_title", empty "steps" and "assessments"
2  foreach step in test_case["steps"] do
3      Retrieve security and expected_response_code for the current step
4      Construct step_details with: purpose, step, expected_response_code, security, and conditional
         outcomes ("if_successful" → "No vulnerability found" and "if_unsuccessful" → "Vulnerability
         found")
5      Append step_details to transformed_case["steps"]
6  end
7  Append phase assessment: "Review outcomes and revisit steps if needed"
8  Add final assessment: "Confirm objectives met"
9  return transformed_case
```

It does this by using a context-sensitive prompt generation algorithm (see Algorithm 3.3) that transforms test cases into a structured format of the Chain-of-Thought prompt. It organizes tasks into sequential phases with steps. In case of the `PENTESTING` context, the steps are defined with the expected results and security conditions, which are used to evaluate the test case. The LLM is prompted with just the steps of the prompt. This can be observed in the example in Figure 3.7, which shows such a prompt. First, the objective is defined as to query the host (**https://reqres.in/api**) and understand its REST API. The second part details the steps, starting with querying the root-level resource endpoints.

Objective: Identify all accessible endpoints via GET requests for **https://reqres.in/api**. ReqRes API is a testing API that allows developers to simulate RESTful interactions.
`"Let's think step by step."` [SIB+24]
Step 1: Query root-level resource endpoints.
Find root-level endpoints for https://reqres.in/api.
Only send GET requests to root-level endpoints with a single path component after the root. This means each path should have exactly one '/' followed by a single word (e.g., '/users', '/products').
1. Send GET requests to new paths only, avoiding any in the lists above.
2. Do not reuse previously tested paths.

Figure 3.7: Example of Chain-of-Thought prompt

Tree-of-Thought

The `TreeOfThoughtPrompt` class implements the Tree-of-Thought (TOT) strategy to generate structured context-sensitive prompts.It is an extension of the `TaskPlanning-Prompt` class and is specifically made to allow for adaptive problem solving and deliberate reasoning for tasks like security analysis and API testing. The class uses a layered structure that changes based on the results at each decision point, which makes it more flexible and responsive. Similar to `ChainOfThoughtPrompt`, for the `PENTESTING` context steps with expected results and security checks to evaluate test cases are defined. The LLM is prompted with these steps only. Figure 3.9 shows an example where the goal is to query the host (**https://reqres.in/api**) and explore its REST API, starting with root-level endpoints.

Algorithm 3.4: Transform Test Case to Tree-of-Thought

```
Input: test_case: Dictionary with fields 'objective', 'steps', 'security',
       'expected_response_code'
purpose: Purpose of the test case
Output: transformed_case: Tree-of-Thought structure
1 Initialize: transformed_case with root objective, empty branches, and assessments
2 foreach step in test_case["steps"] do
3     Retrieve security and expected_response_code
4     Define branch with step details and thoughts (actions and conditions)
5     Append branch to transformed_case["branches"]
6 end
7 return transformed_case
```

`"Imagine three experts each proposing one step at a time. If an expert realizes their step was incorrect, they leave. The question is:"`[Hul23, LZ25]
Objective: Identify all accessible endpoints via GET requests for **https://reqres.in/api**. ReqRes API is a testing API that allows developers to simulate RESTful interactions.
Start by querying root-level resource endpoints.
Focus on sending GET requests only to those endpoints that consist of a single path component directly following the root. For instance, paths should look like '/users' or '/products', with each representing a distinct resource type. Ensure to explore new paths that haven't been previously tested to maximize coverage.

Figure 3.8: Example of Chain-of-Thought prompt

The prompt generation algorithm of `TreeOfThoughtPrompt` (see Algorithm 3.4) uses the TOT strategy by turning test cases into a tree-like structure. It identifies relevant steps based on the prompt's purpose and context, such as documentation or pentesting. Each step is represented as a branch in the tree, with conditional outcomes, assessments, and backtracking paths for iterative reasoning. The resulting structure ensures the execution of the logical and methodical task.

In-Context Learning

The `InContextLearningPrompt` class is designed to generate prompts using the In-Context Learning (ICL) strategy, leveraging historical examples to provide adaptive reasoning and guidance for tasks. Extending the `StatePlanningPrompt` class. The prompts are generated by incorporating prior examples of steps and responses or information from the OpenAPI specification. This allows the prompts to be customized for specific tasks, like exploring API endpoints or checking input parameters, which makes them more accurate and useful. Moreover, the prompts are updated based on past examples and results, which helps improve them over time.

The prompt generation algorithm of `InContextLearningPrompt` (see Algorithm 3.5) adds examples to the prompt to produce example-informed prompts. By retrieving past examples and merging them with current objectives, the algorithm ensures that the prompts are both relevant and instructive. It validates the completeness and clarity of the generated prompts, fostering an iterative learning process. This method does not only enhances adaptive reasoning, but also supports efficient problem solving, particularly in tasks that require detailed exploration and validation.

Algorithm 3.5: Generate In-Context Learning (ICL) Prompt

```
Input: properties: Dictionary of object properties
example_response: Example API response in JSON format
endpoint: Target endpoint of the API
Output: prompt: Generated prompt string
1  Initialize: Start with API core information as prompt
2      Append "REST API: example_response.keys() endpoint", "This API retrieves objects with
         the following properties:"
3      foreach prop, details in properties.items() do
4          Append "- prop:  details['type'] (e.g., details['example'])" to prompt
5      end
6      Append "Example Response:" to prompt
7      if example_response is not empty then
8          Extract example_key as the first key in example_response
9          Format the example_response[example_key] as JSON and append to prompt
10     end
11     return prompt
```

For multistep test cases, the algorithm (see Algorithm 3.6) uses In-Context Learning (ICL) to create prompts. It looks at previous responses and, if a matching example exists, combines it with the current step to build a prompt. If not, it just uses the one from the current step. Each prompt is stored in the `icl_prompts` dictionary based on its purpose. This method helps keep prompts relevant by connecting new tasks with past examples.

```
 Based on this information :
 REST API: dict_keys(['get']) /users
 This API retrieves objects with the following properties:
- id:int (e.g., 1)
- email:str (e.g., george.bluth@reqres.in)
- first_name:str (e.g., George)
- last_name:str (e.g., Bluth)
- avatar:str (e.g., https://reqres.in/img/faces/1-image.jpg)
Example Response:"{
"id": 1,
"email": "george.bluth@reqres.in",
"first_name": "George",
"last_name": "Bluth",
"avatar": "https://reqres.in/img/faces/1-image.jpg"
}" [req23, Cho21]
Objective: Identify all accessible endpoints via GET requests for
https://reqres.in/api. ReqRes API is a testing API that allows developers to
simulate RESTful interactions.
Query root-level resource endpoints.
Find root-level endpoints for https://reqres.in/api.
Only send GET requests to root-level endpoints with a single path component
after the root. This means each path should have exactly one '/'
followed by a single word (e.g., '/users', '/products').
```

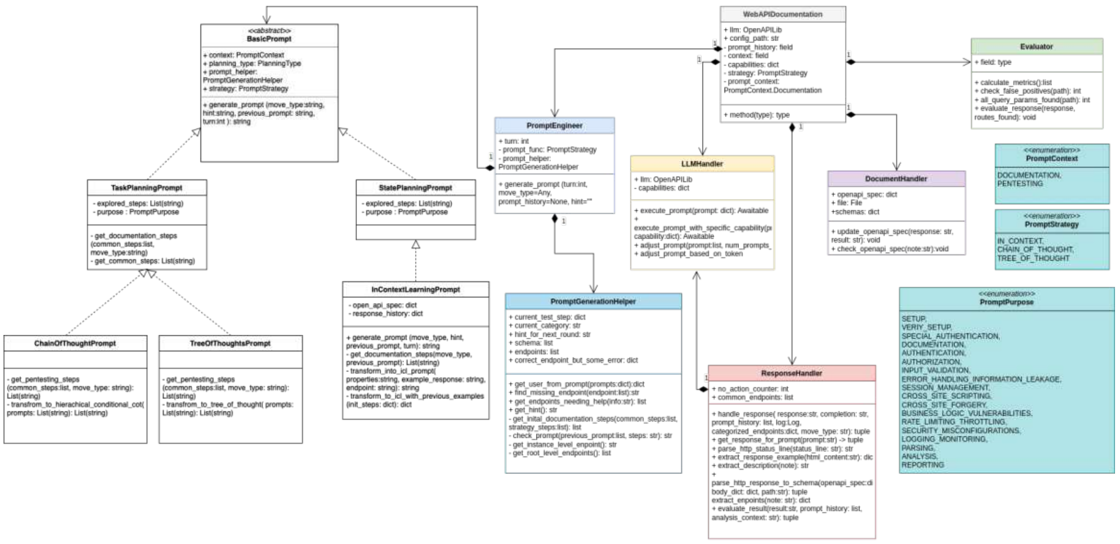
Figure 3.9: Example of Chain-of-Thought prompt, example information taken from [req23, Cho21]

Algorithm 3.6: Transform Steps to In-Context Learning Prompts

```
Input: init_steps: Map of purposes to step groups
Output: icl_prompts: Map of purposes to ICL prompts
1  Initialize: icl_prompts as empty dictionary
2  foreach purpose, steps_groups in init_steps do
3      Retrieve previous_example from response_history
4      foreach steps in steps_groups do
5          foreach step in steps do
6              if previous_example exists then
7                  Format prompt with previous example and current step
8              else
9                  Format prompt using current step only
10             end
11             Append prompt to icl_prompts[purpose]
12         end
13     end
14 end
15 return icl_prompts
```

3.3.2 WebAPIDocumentation

WebAPIDocumentation consists of three phases: **Prompt Generation**, **Documentation**, and **Evaluation** (see Figure 3.10). In the prompt generation phase, prompts are generated based on the chosen strategy and context with the help of the `PromptEngineer`, `PromptGenerationHelper`, and `Prompt` class (see subsection 3.3.1 for further information). In the documentation phase, the responses of the prompts are analyzed via the `ResponseHandler` and documentated via the `DocumentationHandler`.In the evaluation phase, the endpoints and methods found are evaluated against the correct openapi specification. This is done so that it is later possible to compare the effectiveness against other tools.



<<abstract>> BasicPrompt
TaskPlanningPrompt
StatePlanningPrompt
ChainOfThoughtPrompt
TreeOfThoughtsPrompt
InContextLearningPrompt
PromptEngineer
PromptGenerationHelper
WebAPIDocumentation
LLMHandler
DocumentHandler
ResponseHandler
Evaluator
<<enumeration>> PromptContext
DOCUMENTATION, PENTESTING
<<enumeration>> PromptStrategy
IN_CONTEXT, CHAIN_OF_THOUGHT, TREE_OF_THOUGHT
<<enumeration>> PromptPurpose
SETUP, VERIY_SETUP, SPECIAL_AUTHENTICATION, DOCUMENTATION, AUTHENTICATION, AUTHORIZATION, INPUT_VALIDATION, ERROR_HANDLING_INFORMATION_LEAKAGE, SESSION_MANAGEMENT, CROSS_SITE_SCRIPTING, CROSS_SITE_FORGERY, BUSINESS_LOGIC_VULNERABILITIES, RATE_LIMITING_THROTTLING, SECURITY_MISCONFIGURATIONS, LOGGING_MONITORING, PARSING, ANALYSIS, REPORTING

Figure 3.10: Class diagram of WebAPIDocumentation

ResponseHandler

The `ResponseHandler` class analyzes responses, creates schemas, and uses adaptive reasoning. This class simplifies the testing process as it organizes the endpoints and dynamically modifies the testing paths based on past results. It processes HTTP responses, extracts JSON data, and uses LLM-based reasoning to continuously improve its strategies. Moreover, it updates the OpenAPI schemas by extracting critical properties from API responses to accommodate complex data structures.

Moreover, the class detects vulnerabilities through detailed endpoint analysis, automates the creation of OpenAPI specifications from actual API interactions for documentation purposes, and refines testing prompts and strategies based on data-driven insights.

By continually improving its methods based on feedback and incorporating error hints for thorough coverage, the `ResponseHandler` ensures that API testing is systematic and flexible, making it a crucial tool for developers.

OpenAPISpecificationHandler

The `OpenAPISpecificationHandler` class helps to automatically create, manage, and update OpenAPI specifications based on real API use. It uses API responses to ensure that the documentation is accurate and comprehensive. This class dynamically updates OpenAPI documents as it processes API responses, automatically adding details about endpoints, parameters, and schemas. Moreover, it analyzes HTTP responses to organize information and adjusts the documentation by identifying patterns and anomalies in the data.

The `OpenAPISpecificationHandler` starts with an empty OpenAPI template and continually refines it based on new API interactions. Moreover, it automates the documentation process, supports thorough testing, and ensures that all aspects of the API are covered and up-to-date with standards, making API integration smoother.

Evaluator

The `Evaluator` class measures the effectiveness of API documentation and testing. It calculates important metrics, analyzes query parameters, and detect errors such as false positives. By comparing discovered routes and parameters with those documented, it ensures that API testing is complete and documentation accurate.

This class begins by extracting routes and query parameters from API responses and uses this data to track and identify any mismatches or missing information. It calculates metrics such as the coverage of documented routes and parameters and the accuracy of the documentation based on these findings. The `Evaluator` also logs detailed reports, offering insight to improve API testing and documentation strategies.

Ultimately, the `Evaluator` ensures that the API documentation and testing are thorough, improving the overall reliability and effectiveness of the API exploration process.

3.3.3 WebAPITesting

WebAPITesting consists of three phases: **Prompt Generation**, **Response Analysis** and **Test case generation**, and **Evaluation**(see Figure 3.11). In the prompt generation phase, prompts are generated based on context of the `PentestingInformation` and structured in the correct form of prompt strategy (see subsection 3.3.1). In the response analysis and test case generation phase, the responses of the prompts are analyzed via the `ResponseAnalyzer`. Based on this analysis, test cases are generated via the `TestHandler` and the findings are documentated via the `ReportHandler`. In the evaluation phase, the vulnerabilities found are evaluated against the correct openapi specification. This is done so that it is later possible to compare the effectiveness against other tools.

PentestingInformation

The `PenTestingInformation` class helps simplify penetration testing by working with OpenAPI specs to group endpoints and create flexible test steps. It uses an `OpenAPISpecificationParser` to extract endpoints, schemas, and other information from the provided OpenAPI specification. It sets up the tests by creating a user and then logging in. If a token is returned, it is used for further authentication.

In addition to managing and assigning categorized endpoints (public, protected, and sensitive) the class keeps track of accounts and tokens. It also provides built-in steps for security tests, including checks for Authorization, Input Validation, Authentication, Session Management, Error Handling, and testing for vulnerabilities like CSRF, XSS, and business logic errors. For these tests, it creates context-aware prompts based on

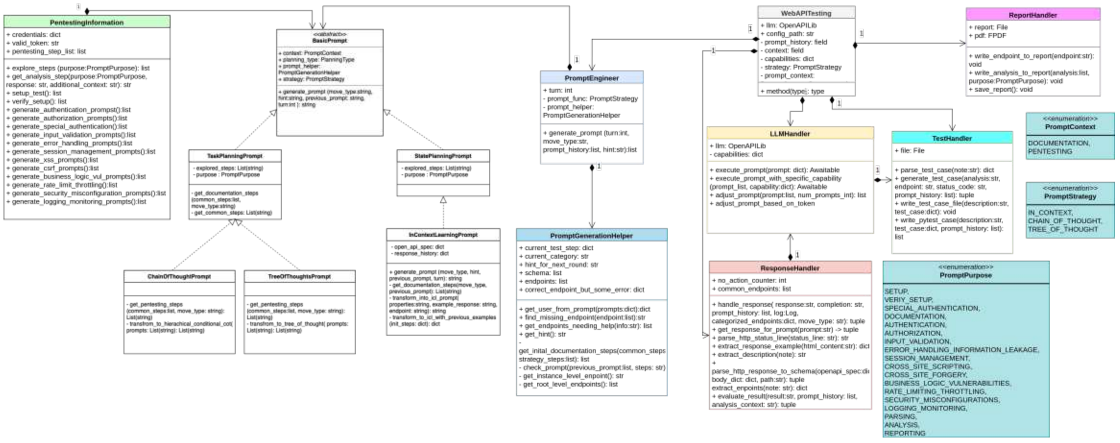

PentestingInformation
<<abstract>> BasicPrompt
PromptEngineer
WebAPITesting
ReportHandler
TaskPlanningPrompt
StatePlanningPrompt
LLMHandler
TestHandler
<<enumeration>> PromptContext
DOCUMENTATION, PENTESTING
<<enumeration>> PromptStrategy
IN_CONTEXT, CHAIN_OF_THOUGHT, TREE_OF_THOUGHT
ChainOfThoughtPrompt
TreeOfThoughtsPrompt
InContextLearningPrompt
PromptGenerationHelper
ResponseHandler
<<enumeration>> PromptPurpose
SETUP, VERIY_SETUP, SPECIAL_AUTHENTICATION, DOCUMENTATION, AUTHENTICATION, AUTHORIZATION, INPUT_VALIDATION, ERROR_HANDLING_INFORMATION_LEAKAGE, SESSION_MANAGEMENT, CROSS_SITE_SCRIPTING, CROSS_SITE_FORGERY, BUSINESS_LOGIC_VULNERABILITIES, RATE_LIMITING_THROTTLING, SECURITY_MISCONFIGURATIONS, LOGGING_MONITORING, PARSING, ANALYSIS, REPORTING

Figure 3.11: Class diagram of WebAPITesting

user-defined purposes, example data, and endpoint-specific schemas. Moreover, the class includes functions for validating security configurations, testing edge cases, and investigating endpoint behavior in various contexts.

Overall, the `PenTestingInformation` class helps find API vulnerabilities and misconfiguration, makes penetration testing easier and faster by automating the process and reducing manual work.

ResponseAnalyzer

The `ResponseAnalyzer` class analyzes the HTTP responses to check for issues in areas like Authentication, Authorization, and Input validation. Depending on its goal, it can also check for things like missing security headers or invalid inputs. It breaks down raw HTTP responses into components like status codes, headers, and bodies to enable precise, goal-oriented analysis. Moreover, The class adjusts its analysis based on the goal, allowing it to perform specific checks like confirming if authentication worked or if the input is valid. It looks for important headers, checks for rate limits, and makes sure security headers are included.

Overall, `ResponseAnalyzer` works by reading HTTP responses, checking them based on their purpose, and recording the results. This clear process helps spot issues and gives useful feedback to improve how the API works and how secure it is.

TestHandler

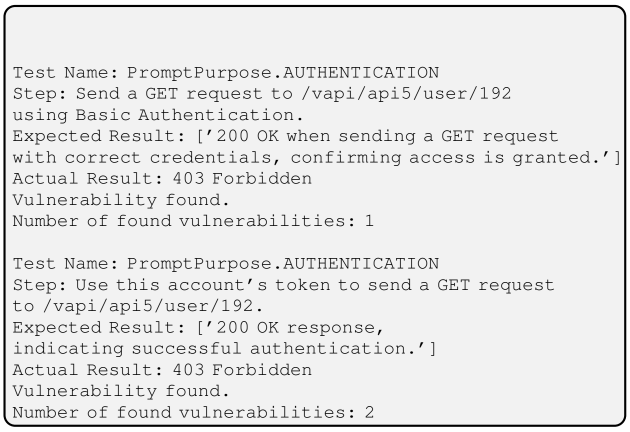
The `TestHandler` class automates the creation, management, and execution of test cases for API endpoints. The class uses structured parsing and LLMs to generate test cases scripts that work with `pytest` , which makes the API testing process much easier and faster.It pulls key details from response descriptions, like HTTP methods, endpoints, input data, and expected results. After that it uses the LLM to create test cases with the right checks and expected outcomes. These test cases are saved in organized files, and Python test scripts are created to run smoothly with pytest.

The workflow includes reading test case descriptions, generating test cases with AI, saving them to files, converting them into pytest scripts, and managing these scripts to make sure they are correct and reusable.

Overall, the TestHandler class improves the accuracy and speed of API testing while reducing the amount of manual work needed. For this, it uses structured parsing and LLMs to generate detailed test cases and scripts compatible with `pytest`, which significantly streamlines the API testing process.

ReportHandler

The `ReportHandler` class makes the creation and organization of detailed reports for API testing easier. It supports both text and PDF formats to capture and organize essential information effectively (see Figure 3.12).



```
Test Name: PromptPurpose.AUTHENTICATION
Step: Send a GET request to /vapi/api5/user/192
using Basic Authentication.
Expected Result: ['200 OK when sending a GET request
with correct credentials, confirming access is granted.']
Actual Result: 403 Forbidden
Vulnerability found.
Number of found vulnerabilities: 1

Test Name: PromptPurpose.AUTHENTICATION
Step: Use this account's token to send a GET request
to /vapi/api5/user/192.
Expected Result: ['200 OK response,
indicating successful authentication.']
Actual Result: 403 Forbidden
Vulnerability found.
Number of found vulnerabilities: 2
```

Figure 3.12: Passage of Vulnerability Report

The class automatically handles the creation and management of unique report files for each testing session, which ensures that no data is overwritten. It comprehensively documents the details of the endpoints tested and their results, along with analysis results that include insights and objectives.

The workflow of the `ReportHandler` includes setting up a unique report file, logging details of the endpoints tested, categorizing and writing the analysis results according to their purposes, and finalizing the report in PDF format for easy distribution and archival.

Overall, the `ReportHandler` class simplifies the generation of professional, detailed reports, which is improtant for the reproducibility of the tests.

3.3.4 Summary

This thesis was implemented in the "web_api_testing" use case of the open-source project *HackingBuddyGPT*[3]. *HackingBuddyGPT* [HC23] is a tool powered by LLMs that helps with ethical hacking and API security testing. It automates tasks like finding vulnerabilities, running penetration tests, and creating test cases. It addresses the OWASP Top 10 risks and uses LLMs to simplify workflows and improve efficiency.

At the core of *HackingBuddyGPT* is the `PromptEngineer`, which generates adaptive context-aware prompts using strategies such as `CHAIN_OF_THOUGHT`, `TREE_OF_THOUGHT`, or `IN_CONTEXT_LEARNING` for efficient endpoint discovery and vulnerability detection. The `PromptGenerationHelper` helps by reviewing prompts, tracking progress, and filling in missing parts of the API schema.

The `PenTestingInformation` class is a central component for penetration testing, integrating OpenAPI specifications to dynamically categorize endpoints ("public", "protected", "sensitive") and generate customized testing steps. Moreover, it manages accounts, tokens, and runs common security tests like Authentication, Authorization, and checks for issues like XSS, CSRF, and business logic flaws to ensure full testing coverage.

Other important parts of the system are the **ResponseHandler** and **OpenAPISpecificationHandler**, which handle live response analysis and automatic API documentation. The TestHandler creates test cases automatically, while the ReportHandler generates detailed reports for sharing and documentation.

By combining smart prompt strategies, automated testing, and clear documentation, HackingBuddyGPT offers a strong and efficient tool for API security testing, helping uncover vulnerabilities and deliver useful insights.

[3]`https://github.com/ipa-lab/hackingBuddyGPT`

CHAPTER 4

Evaluation

The objectives outlined in Section 3 will be evaluated using the metrics listed in Section 4.1.1, and compared against the benchmarks provided in Table 4.1 and Table 4.2. While the documentation metrics are derived from related works, the pentesting metrics are based on previous research by Isozaki et al. [ISCK24]. To answer the question of how effective different Large Language Models (LLMs) are at generating test cases for REST APIs and what factors influence their performance, the following experiments were conducted with two LLMs: GPT-4o-mini and o1. The results of these experiments will be described and analyzed in this section. Although both models are developed by OpenAI, GPT-4o-mini is more versatile and designed for broad applications, while o1 is optimized for specific tasks, which can enhance certain performance aspects. This comparison looks at how well each model handles different tasks, showing their strengths and weaknesses, and giving insight into how their design affects their real-world performance.

4.1 Methodology

To evaluate our approach, we used a three-step process: setup, execution, and analysis. This helped us keep things consistent across different tests and ensured our results could be repeated. Our method combines ideas from both documentation-based and security-focused testing to measure how accurate and effective the system is.

We tested a wide range of APIs, which we grouped into **documentation benchmarks** and **pentesting benchmarks** (Table 4.1 and Table 4.2). The documentation benchmarks, inspired by works like RESTSpecIT, test route and parameter discovery. In contrast, the pentesting benchmarks include OWASP benchmarks and a custom "ticketbuddy" benchmark for security testing, enabling a comprehensive assessment of functional and security aspects.

The evaluation process includes the following.

1. **Benchmark Preparation:** Making sure each API benchmark had working access, specs, and needed tools.
2. **Metrics Definition:** Metrics include % Routes Found, % Parameters Found, % Vulnerabilities Found, and Number of Steps.
3. **Execution:** Each benchmark was tested 10 times to ensure consistent results.
4. **Data Collection:** We recorded key values like routes discovered and vulnerabilities detected.
5. **Analysis:** We studied the data to find what worked well and what did not. For the pentesting benchmarks, we also reviewed the generated test reports.

For the documentation benchmarks, we focused on finding routes and parameters accurately. While for the pentesting benchmarks, our goal was to detect common security flaws like injection attacks and broken authentication. By looking at both sets together, we got a clear and consistent view of how well our method works compared to other tools.

4.1.1 Metrics

The documentation metrics focus on the percentage of routes and parameters accurately identified, as well as the discovery of additional routes and parameters. The Pentesting metrics include the percentage of vulnerabilities found, execution time and workload, as derived from Isozaki et al. [ISCK24].

4.1.2 Documentation

For documentation benchmarks, we evaluated the ability to accurately identify and document routes and parameters. As our baseline we used the results of RESTSpecIt.

- **% Routes Found:** This measurement is measured over 10 runs and denotes the average percentage of correctly identified routes.
- **% Parameters Found:** This measurement is measured over 10 runs and denotes the average percentage of correctly identified parameters.
- **# Additional Routes Found:** This measurement denotes the average number of additional routes found over 10 runs, and is measured relative to the baseline set of routes initially identified by RESTSpecIT.
- **Number of steps for discovery:** This measurement is measured over 10 runs and denotes the average number of steps until all endpoints and parameters are found.

To thoroughly evaluate the capability of our framework to discover and document both routes and query parameters, we used a diverse set of REST APIs. Table 4.1 presents the APIs we selected, grouped by complexity. We classify REST APIs into different groups:

API	Type	Description	Use Cases	URL
An API of Ice and Fire	Simple	API that proviedes data from the "A Song of Ice and Fire" series.	Testing route discovery, parameter extraction, data retrieval.	`https://anapioficeandfire.com/`, Last accessed 2025-04-02
BallDontLie NBA	Simple	A free NBA API that provides real-time and historical basketball data.	Real-time data extraction, route discovery, parameter identification.	`https://balldontlie.io/`, Last accessed 2025-04-02
Random User Generator	Medium	API that generates random user data like names, emails, addresses.	Dynamic data retrieval, parameter handling, route discovery.	`https://randomuser.me/`, Last accessed 2025-04-02
CoinCap	Hard	API that containsReal-time cryptocurrency market data.	Real-time data streaming, route discovery, parameter extraction.	`https://coincap.io/`
Open Brewery DB	Hard	API that includes information about breweries, names, types, locations.	Route discovery, parameter handling, location-based queries.	`https://www.openbrewerydb.org/`, Last accessed 2025-04-02
ReqRes	Hard	Mock data for testing and prototyping, supports CRUD operations.	API route discovery, CRUD operations, parameter handling.	`https://reqres.in/`, Last accessed 2025-04-02

Table 4.1: Documentation Benchmarks

- **Simple**: They are small, well-defined endpoint sets and limited parameters. These ensure that our framework can handle relatively straightforward documentation tasks without excessive overhead.
- **Medium**: They have a small set of endpoints but a large set of query parameters. (e.g, Random User Generator)
- **Hard**: These APIs feature extensive endpoints, intricate parameter structures, real-time data streaming, and CRUD operations(e.g., CoinCap, GBIF Species, Open Brewery DB, ReqRes).

We chose a variety of complex APIs on purpose to show that the system can do more than just find routes. Moreover, it handles real-time data, structured data, and advanced query parameters well. We included harder REST APIs to test if our framework can scale. By using both simple and complex APIs, we show that the framework is flexible and performs well in real-world situations.

4.1.3 Pentesting

The metrics used in pentesting provide insight into the efficiency and resource demands of the testing framework.

- **Precision**: Itevaluates how accurately positive predictions are made, calculated as[HR24]:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} [\text{HR24}] \tag{4.1}$$

A high precision rate means that there are fewer false positives, which means that the system correctly identifies relevant items while minimizing incorrect detections[HR24]. In API security testing, high precision ensures that detected vulnerabilities are real threats rather than false alarms, reducing unnecessary manual verification.

API	Description	Use Cases	URL
TicketBuddy	Custom benchmark API for security testing, includes SQL Injection, Identification and Authentication Failures, and more.	Testing vulnerabilities in Access Control, Data Validation, Authentication methods, and Injection attacks.	https://github.com/Neverbolt/ticketbuddy, Last accessed 2025-03-15
OWASP vAPI	Deliberately vulnerable API for security testing, includes Injection Flaws, Identification and Authentication Failures, and improper data validation.	Testing Injection attacks, authentication mechanisms, data validation, and access control vulnerabilities.	https://owasp.org/www-project-vAPI/, Last accessed 2025-04-02
OWASP crAPI	Vulnerable API designed to simulate real-world security issues, includes Identification and Authentication Failures, sensitive data exposure, and IDOR.	Testing access control flaws, sensitive data exposure, IDOR, and authentication vulnerabilities.	https://owasp.org/www-project-crAPI/, Last accessed 2025-04-02
OWASP Juice Shop	Insecure web application for security testing, includes SQL Injection, XSS, and authentication flaws.	Testing authentication and authorization mechanisms, input validation, SQL Injection, XSS, and business logic vulnerabilities.	https://owasp.org/www-project-juice-shop/, Last accessed 2025-04-02

Table 4.2: Pentesting Benchmarks

The vulnerabilities in Table 4.3 come from a custom benchmark called TicketBuddy, which was created to test for issues that can be found through REST API calls. These vulnerabilities include Function Level Authorization, Broken Access Control, Lack of Resources and Rate Limiting, Broken , Identification and Authentication Failures, Security Misconfiguration, and Injection (SQL, NoSQL, etc.). The goal of TicketBuddy is to focus on common API security issues that can be identified through interactions with the API itself. It is important to find these vulnerabilities during REST API testing because they can harm the system's security.

No.	Vulnerability	How to Find the Vulnerability	Detectable?
1	Broken Access Control	Tries to make an API request to access other users' data.	✓
2	Identification and Authentication Failures	Tests the authentication mechanism by attempting brute-force or credential stuffing attacks using weak, default, or leaked passwords. Use tools like Burp Suite for automated attacks.	✓
3	Lack of Resources and Rate Limiting	Performs a DoS attack or send excessive requests to see if the API limits or blocks excessive calls. A lack of rate-limiting can lead to brute-force or DoS vulnerabilities.	✓
4	Broken Function Level Authorization	Tries accessing higher-privileged API functions by modifying the endpoint or parameters (e.g., accessing admin-only functions) to check if proper authorization is enforced.	✓
5	Security Misconfiguration	Reviews the API's security configuration for common misconfigurations, such as improper HTTP headers, exposed services, verbose error messages, or default credentials.	✓
6	Injection (SQL, etc.)	Tries injecting malicious input (e.g., SQL, NoSQL) into API parameters or request bodies. Tools like Burp Suite Intruder can automate testing for Injection Flaws.	✓

Table 4.3: List of Vulnerabilities in TicketBuddy that can be found via REST API Testing [HSAB22]

Each benchmark API provides a unique combination of vulnerabilities that are crucial for evaluating pentesting frameworks based on the described metrics. **TicketBuddy** includes common issues like SQL Injection, Authentication flaws, and Sensitive data leaks, which makes it useful for testing how well a framework can detect different types of vulnerabilities. Another benchmark is **OWASP vAPI** , which is a purposely vulnerable API that is great for checking both how fast (execution time) and how thoroughly (% of Vulnerabilities Found) a framework can spot basic attack patterns. **OWASP Juice Shop** is known for a wide range of bugs, including XSS and business logic issues. It's especially helpful for testing how well a tool handles complex API flows and performance under pressure.

No.	Vulnerability	How to Find the Vulnerability	Detectable?
1	Broken Access Control	Intercepts the API request and try modifying the object ID in the request parameters (e.g., change 'id=5' to 'id=6') to access other users' data.	✓
2	Identification and Authentication Failures	Tests the authentication mechanism by attempting brute-force or credential stuffing attacks using weak, default, or leaked passwords. Use creds.csv file to obtain user data.	✗
3	Excessive Data Exposure	Uses a proxy tool such as Burp Suite to inspect the response from the API and check if sensitive data is exposed that should be filtered out on the client-side.	✗
4	Lack of Resources and Rate Limiting	Performs a DoS attack or send excessive requests to see if the API limits or blocks excessive calls. User should get 4 digit code to phone.	✗
5	Broken Function Level Authorization	Tries accessing higher-privileged API functions by modifying the endpoint or parameters (e.g., accessing admin-only functions) to check if proper authorization is enforced.	✓
6	Unrestricted Access to Sensitive Business Flows	Checks for endpoints that accept user input and try appending unexpected fields, like '&admin=true' in the request body, to gain unauthorized access to restricted functionality.	✗
7	Security Misconfiguration	Reviews the API's security configuration for common misconfigurations, such as improper HTTP headers, exposed services, verbose error messages, or default credentials.	✓
8	Injection (SQL, etc.)	Tries injecting malicious input (e.g., SQL, NoSQL) into API parameters or request bodies. Tools like Burp Suite Intruder can automate testing for Injection Flaws.	✓
9	Improper Asset Management	Enumerates sold or unused API versions that might be exposed unintentionally and looks for deprecated endpoints or API versions that might lack security updates.	✗
10	Insufficient Logging and Monitoring	Tests by performing suspicious activities and checking if the API logs or alerts any of those actions. This vulnerability allows attackers to act unnoticed.	✗

Table 4.4: Vulnerabilities in vAPI that can be found via REST API Testing [HSAB22]

Table 4.4 lists the OWASP API Top 10 vulnerabilities found in vAPI, how they can be detected, and whether they can be found using REST API testing. This thesis focuses on vulnerabilities that can be only found through REST API calls. These include Injection Flaws (SQL/NoSQL), Mass Assignment, Lack of Rate Limiting, Broken Object-Level Authorization, Identification and Authentication Failures, and Security Misconfigurations. Direct testing of these problems is possible by interacting with the API. Some vulnerabilities like Excessive Data Exposure, poor asset tracking, or missing logs cannot be tested directly through API calls, therefore, they are not included in this testing procedure. This keeps the focus on security issues that can actually be found using standard API testing methods.

Table 4.5 shows the security issues that can be found in the crAPI benchmark using REST API testing like Excessive Data Exposure, Broken Function-Level Authorization, SQL/NoSQL Injection, and Unauthenticated Access. Testers can uncover these issues by tweaking request parameters, reviewing responses, or trying unauthorized requests. Since they relate to how the API handles input, permissions, and database queries, they're

No.	Vulnerability	How to Find the Vulnerability	Detectable?
1	BOLA (Broken Object-Level Authorization)	Intercepts the "Refresh Location" request in Burp Suite, modify the vehicle ID from another user's ID, and send the request to retrieve unauthorized vehicle data.	✗
2	Broken User Authentication	Uses Burp Suite's Intruder to automate OTP attempts with incorrect values on the "Forgot Password" page, using an unprotected version of the endpoint (v2).	✗
3	Excessive Data Exposure	Inspects responses from the "Community" page for sensitive information like another user's details exposed in API responses.	✓
4	Rate Limiting	Modifies parameters like "repeat_request_if_failed" and "number_of_repeats" to perform a Layer 7 DoS attack on the "Contact Mechanic" feature, causing server overload.	✗
5	Broken Function Level Authorization	Changes video name using PUT, discover DELETE method available, and attempts to delete videos from other users by modifying video identifiers.	✓
6	Mass Assignment	Modify the "status" and "quantity" parameters in API requests to return items or increase balances, such as using PUT instead of GET to alter order status.	✗
7	Server-Side Request Forgery (SSRF)	Intercept a form request and change the URL to an external site (e.g., www.google.com) to trigger a server-side HTTP request.	✗
8	NoSQL Injection	Intercepts and modifies the coupon validation request with NoSQL payloads (e.g., ‘"$ne": 1‘) to gain unauthorized access.	✗
9	SQL Injection	Inject SQL queries (e.g., ‘0’ OR ’0’ = ’0‘) into the coupon redemption process to manipulate the database and redeem coupons.	✓
10	Unauthenticated Access	Sends an unauthenticated request to an endpoint that provides data (e.g., order details) without requiring authentication.	✓
11	JWT Token Forgery	Uses Burp Suite's JWT Editor to modify JWT token's algorithm to "none" and changes the user email to access another user's information.	✗

Table 4.5: List of Vulnerabilities in crAPI that can be found via REST API Testing [HSAB22]

well-suited for standard API testing.

For instance, user roles and permissions must be examined for Broken Object-Level Authorization, which isn't always apparent through API requests. Similarly, Broken User Authentication depends on the behavior of the authentication system, which may not be observable via API calls. For this REST API, it is only possible to find this vulnerability by obtaining the creds.csv file from the GitHub repository. Moreover, Rate Limiting problems are hard to find with REST API testing alone because they depend on server-side configurations that are hidden in API responses. Some parts of Server-Side Request Forgery (SSRF) can be tested through REST API calls, but fully understanding its impact requires watching how the server behaves, since SSRF makes the server request internal resources. Another vulnerability that is hard to test with regular REST API calls is JWT Token Forgery, since it requires changing tokens and knowing how the server verifies them. Because these issues depend on server-side logic, REST API testing alone often isn't enough to catch them.

The Table 4.14 lists vulnerabilities in the OWASP Juice Shop and indicates whether they can be detected via REST API testing. Vulnerabilities like Authentication problems, Sensitive Data Leaks, Broken Access Control, and NoSQL injection can be found by testing the REST API. These issues come from how the app handles data, checks users, and runs database queries. Some other vulnerabilities, marked with "✗" in the last column, cannot be found just by using REST API calls. For example, while it is possible to send a SQL command in an REST API request, some SQL injection issues only show up if the database actually runs the query. Cross-Site Scripting (XSS) is another vulnerability

No.	Vulnerability	How to Find the Vulnerability	Detectable?
1	Injection (SQL, Command, Email)	Intercept API requests and inject payloads like “ OR 1=1 –‘ in input fields or query parameters to manipulate database operations.	✗
2	Identification and Authentication Failures	Tests the login and password reset endpoints with weak or brute-force attempts to exploit authentication mechanisms.	✓
3	Sensitive Data Exposure	Inspects API responses for unencrypted sensitive data such as credit card numbers or personal information.	✓
4	Broken Access Control	Modifies parameters like user IDs in API requests (e.g., change ‘/rest/basket/1‘ to ‘/rest/basket/2‘) to access unauthorized data.	✓
5	Cross-Site Scripting (XSS)	Send payloads such as ‘<script>alert(1)</script>‘ via API inputs and check if the response reflects it unsanitized.	✗
6	Insufficient Rate Limiting	Makes repeated requests to endpoints (e.g., login, registration) and check if the system limits the number of attempts.	✓
7	Server-Side Request Forgery (SSRF)	Modifies parameters of the URL in API requests to try to access internal or unauthorized resources.	✗
8	Mass Assignment	Manipulates input fields by sending extra fields or modifying object properties through API calls (e.g., updating user roles).	✓
9	NoSQL Injection	Injects NoSQL payloads (e.g., ‘"$ne": 1‘) in the API request to manipulate database queries.	✓
10	Information Disclosure via Headers	Inspect HTTP headers from API responses for sensitive data such as server details or API version information.	✓

Table 4.6: OWASP Juice Shop Vulnerabilities that can be found via REST API Testing [HSAB22]

that cannot be tested vie REST API calls. XSS is a client-side issue that happens when the browser runs unsafe scripts. Another similar vulnerability is Server-Side Request Forgery (SSRF), which occurs when the server is tricked into making requests to internal or unauthorized resources. The effects of SSRF usually show up on the server, not in the API response. Therefore, SSRF vulnerabilities require testing methods that involve the server's processing of requests, which cannot be fully assessed using only API calls. These vulnerabilities require additional testing approaches beyond REST API interactions to be detected effectively.

4.1.4 Summary

This thesis introduces two new use cases for HackingBuddyGPT: WebAPIDocumentation and WebAPITesting. WebAPIDocumentation generates an OpenAPI specification, which ensures that there is no missing data. WebAPITesting checks if the REST APIs work correctly and are secure by using the OWASP Top 10 as a guideline. It runs tests to find common problems like SQL injection or broken authentication. Together, these tools help create better test cases and let us compare how well different frameworks find vulnerabilities and document APIs.

4.2 Results: WebAPIDocumentation

In this section, the results are presented and analyzed. Each experiment was done with GPT-4o-mini and o1.

WebAPIDocumentation was run 10 times on each of the REST APIs on all types of prompt strategies (COT, TOT, ICL) so that it is comparable to the results of RESTSpecIT.

4.2.1 OpenAI GPT-4o-mini

The experiments in this section were performed using GPT-4o-mini. The table provided in Table 4.7 offers a comparative analysis of our framework HackingBuddyGPT and the already established RESTSpecIT, focusing on their ability to discover API routes and parameters across a selection of APIs. The table (see Table 4.7) shows two main metrics: the percentage of routes found and the percentage of parameters found. Results for each tool are shown side by side for easy comparison.

API name	**% Found Routes**		**% Found Parameters**	
	HackingBuddyGPT	RESTSpecIT	HackingBuddyGPT	RESTSpecIT
An API of Ice and Fire	100%	98.57%	100%	93.53%
BALLDONTLIE NBA	100%	96.25%	71.43%	90.83%
Random User Generator	100%	100.00%	100%	73.33%
CoinCap	100%	77.00%	71.43%	80.66%
Open Brewery DB	100%	61.67%	100%	79.23%
ReqRes	100%	87.50%	66.67%	100.00%

Table 4.7: Comparison of best HackingBuddyGPT and RESTSpecIT results based on GPT-4o-mini on APIs and evaluation metrics

Route discovery When evaluating the performance of HackingBuddyGPT and RESTSpecIT in discovering API endpoints (routes), it becomes apparent from Table 4.7 that HackingBuddyGPT generally outperforms RESTSpecIT. For both **simple** REST APIs, both frameworks perform very well, which can be attributed to the low number of endpoints that have to be discovered. It even manages 100% of the query parameters for the *An API of Ice and Fire* API.

For the **medium** REST API *Random User Generator*, both frameworks achieve 100% endpoint discovery. This could be due to the simplicity of this REST API since only has one endpoint called "/api."

Moreover, HackingBuddyGPT performed well in endpoint discovery for **hard** REST APIs, achieving 100% for all such APIs. In contrast, RESTSpecIT was only able to achieve scores between 60% and 90% in endpoint discovery for these REST APIs. The best performance of RESTSpecIT was achieved for *ReqRes* with 87.50% in endpoint discovery. In addition, it fell notably short in APIs like *Open Brewery DB*, where only 61.67% of the routes were discovered, which is surprising, since this REST API only has four endpoints, which are variants of breweries" such as /breweries/search." HackingBuddyGPT likely performs better in route discovery for APIs like *Open Brewery DB* because it focuses on common endpoint patterns during exploration. This approach seems to help it find more routes. This can also be observed for the *CoinCap* benchmark, where our approach found all the routes (100%), while RESTSpecIT found only 77%. This implies that the route discovery methodology used by HackingBuddyGPT might be more thorough or more in line with the architecture of specific APIs. This suggests that HackingBuddyGPT's method is either more thorough or better matched to how some APIs are built. It's also worth noting that HackingBuddyGPT found 100

Parameter discovery Both tools show varying effectiveness. For instance, HackingBuddyGPT and RESTSpecIT both scored highly on the two **simple** REST APIs (*An API of Ice and Fire* and *Random User Generator*). However, HackingBuddyGPT struggles with identifying more complex parameters in **hard** REST APIs like CoinCap and Open Brewery DB. Moreover, it has difficulty finding all parameters in *BALLDONTLIE NBA*, which can be due to specific endpoint requirements. For instance, the '/seasons_averages' endpoint requires both a category in the route and an additional type as a query parameter, which cannot be retrieved with GET requests. This makes the discovery more challenging. The better performance of RESTSpecIT in terms of parameter discovery for these REST APIs could be due to its method of discovering query parameters. In contrast, HackingBuddyGPT uses the LLM to create parameters using descriptions of the REST API along with dynamically generated hints, making the inference process easier for the LLM.This could be due to the fact that these REST APIs use a lot of uncommon parameters that are harder to infer than the others. RESTSpecIT explores REST APIs by mutating and inferring HTTP requests through a masking strategy. Using route mutations, it adds, removes, or modifies paths to uncover undocumented endpoints. Query parameter mutations introduce, alter, or reset parameters to test API flexibility, while value mutations generate diverse inputs based on randomization, boundary values, and inferred data types. For example, it might modify a URL path, try new query options, or send unusual input values like long strings or edge-case numbers. Each successful request is saved and used as a starting point for more testing. This lets RESTSpecIT explore APIs in a smarter way than standard black-box tools, helping it uncover hidden endpoints and behaviors. Each successful mutation is stored as a seed for further exploration, which allows RESTSpecIT to iteratively refine its understanding and maximize API coverage beyond traditional black-box testing methods. This method of discovering query parameters performs better for the **medium** type REST API (e.g., *Random User Generator*), where HackingBuddyGPT achieved 100% for the query parameters, whereas RESTSpecIT only discovered 73.33%.

For *BALLDONTLIE NBA*, HackingBuddyGPT achieved 71.43% for query parameter discovery, while RESTSpecIT achieved 90.83%. For *CoinCap*, HackingBuddyGPT achieved 71.43% for parameter discovery, while RESTSpecIT achieved 80.66%. RESTSpecIT only achieved 79.23% in parameter discovery for *Open Brewery DB*, whereas HackingBuddyGPT achieved 100%. However, RESTSpecIT achieved 100% in parameter discovery for *ReqRes*, whereas HackingBuddyGPT merely achieved 66.67%.

This experiments show that HackingBuddyGPT is better at route discovery, particularly for **simple** and **hard** REST APIs, while RESTSpecIT is better at parameter discovery, particularly for **medium** and **hard** REST APIs. Since each tool has benefits, selecting one over the other should depend on the specific needs of the API and the type of data sought.

API	% Found Routes			% Found Parameters			# Additional Found Routes		
	COT	TOT	ICL	COT	TOT	ICL	COT	TOT	ICL
An API of Ice and Fire	100%	83.33%	100%	100%	100%	100%	0	0	0
BALLDONTLIE NBA	100%	75%	75%	71.43%	37.50%	37.5%	0	0	0
Random User Generator	100%	100%	100%	100%	100%	100%	2	2	2
CoinCap	100%	80%	80%	71.43%	71.88%	57.14%	1	2	1
Open Brewery DB	100%	100%	100%	100%	100%	100%	0	0	0
ReqRes	100%	100%	100%	42.86%	42.86%	66.67%	45	96	108

Table 4.8: Average Evaluation Metrics for HackingBuddyGPT Across APIs over 10 runs with GPT-4o-mini

Effect of Prompting Strategies The table in Figure 4.8 provides a detailed comparison of the performance of the different Prompt Engineering strategies in various APIs, revealing that while endpoint discovery is nearly perfect, with all prompt strategies consistently achieving close to 100% in the tested APIs, parameter discovery shows more variability and generally lower success rates. The Chain-of-Thought (COT) method outperforms the other methods in the *CoinCap* API when it comes to identifying parameters. This suggests that its methodical, sequential approach to reasoning may be better suited for intricate parameter configurations. It's possible that there are several reasons why the In-Context-Learning method performs the worst when it comes to parameter discovery. This method's primary limitation is that it depends on the context of the original input, which may not always contain enough relevant details to disclose more nuanced or hidden parameters. Additionally, In-Context-Learning might not be as effective at adapting dynamically to the different structures and conventions used across different APIs as methods that explicitly analyze or change requests and responses to find parameters.

The COT and ICL strategies achieve 100% in both route and parameter discovery for simpler APIs such as *An API of Ice and Fire*, while the TOT strategies performs marginally worse, achieving 83.33% in route discovery. This shows that the simple design of this API enables HackingBuddyGPT and the other frameworks to operate as efficiently as possible without finding new paths.

All strategies perform well in route discovery for the more complex *BALLDONTLIE NBA* API, with COT reaching 100%. However, there is a noticeable difference in parameter discovery. In contrast to the TOT strategy, which finds 37.50% of the parameters, COT finds 71.43%. This implies the more methodical approach of COT is more successful in this case when it comes to establishing parameters. Additionally, no strategy finds any new routes, showing that all tools successfully control the relatively fixed structure of the API.

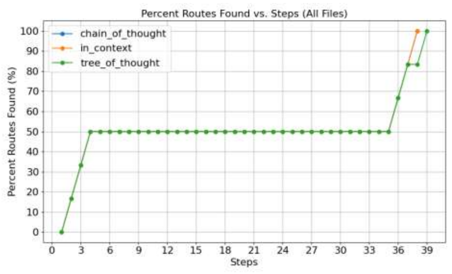
As expected for the simple API *Random User Generator*, HackingBuddyGPT identified routes and parameters with perfect accuracy (100%) across all strategies. Both tools can explore other or different structures in this API, as evidenced by the two extra routes that are found by all strategies.

While COT is 100% successful in route discovery for *CoinCap*, it only detects 71.43% of the parameters in parameter discovery. In comparison to the other strategies, the TOT and ICL strategies find a few more routes and perform marginally better in parameter discovery (71.88% and 57.14%, respectively). In particular, TOT and ICL each find two extra routes.

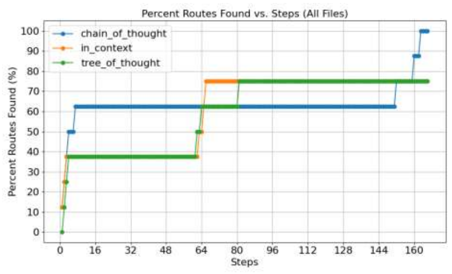
The *Open Brewery DB* API presents another straightforward example where all strategies perform similarly well. All strategies achieve 100% both in the discovery of routes and parameters, finding no additional routes. This is probably due to the API's clear structure, which eliminates the need for further exploration.

On the other hand, *ReqRes* turns out to be the hardest API. While HackingBuddyGPT achieves 100% in route discovery, it performs worse in parameter discovery, with ICL achieving 66.67% and COT and TOT identifying 42.86% of the parameters. Regarding extra routes, the COT strategy discovers 45, TOT discovers 96, and ICL discovers 108.

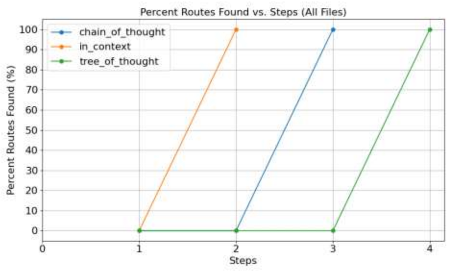
This analysis highlights the importance of choosing a strategy based on the desired type of information and API specifics, indicating the need for enhanced adaptive strategies for parameter discovery in automated API testing tools.



(a) An API of Fire and Ice



(b) BALLDONTLIE NBA

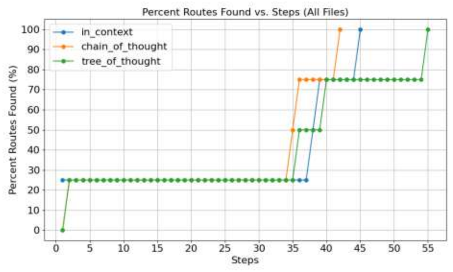


(c) Random User Generator

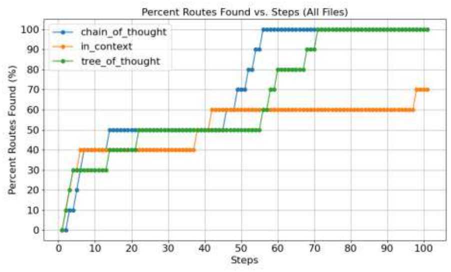
Figure 4.1: Number of steps in route discovery for GPT-4o-mini for simple and medium benchmarks

Effect of Prompt Stratgies Figure 4.1 presents the results of the route discovery strategies in five APIs using the GPT-4o-mini model. In*An API of Fire and Ice* (a), all strategies reached 100% route discovery fairly quickly. Chain-of-Thought and Tree-of-Thought finished by step 39, while In-Context Learning got there a bit earlier — likely because the API was simple to explore. In contrast, with the *BALLDONTLIE NBA* (b), Chain-of-Thought performed the best. It reached 60% after just a few steps, briefly plateaued there and then continued to 100%, while the other two Prompt Engineering strategies struggled to exceed 75%. Chain-of-Thought may have done better because it breaks the problem into smaller steps, which seems to help it get more accurate results and reach the goal faster, especially with complex or less straightforward structures.

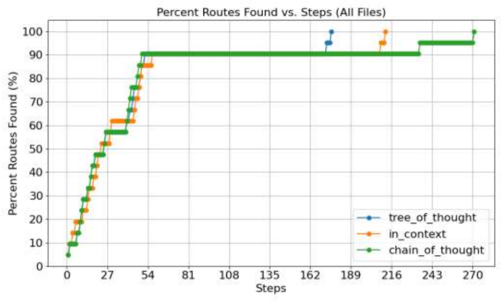
In *Random User Generator* (c), In-Context Learning excelled, finding everything in the first step and clearly outperforming the other strategies. This shows that it works well for simpler API structures.



(a) Open Brewery DB



(b) CoinCap



(c) Reqres

Figure 4.2: Number of steps in route discovery for GPT-4o-mini for hard benchmarks

In *Open Brewery DB* (see Figure 4.2 (a), Chain-of-Thought and In-Context strategies reached 100% very quickly. Tree-of-Thought was slower but caught up by step 55, which suggests taht the REST API had moderate complexity. The *CoinCap* API (b) showed varied performance between strategies. Tree-of-Thought and Chain-of-Thought reached full discovery by the 70th step, while In-Context learning plateaued at 80% after 97 steps. This suggests that In-Context learning struggled with the complexity of this API or had difficulty adapting to its structure. The most challenging API appeared to be *Reqres* (c), where all strategies required many steps to reach 100% discovery, indicating a highly complex API or suboptimal documentation.

Overall, Tree-of-Thought demonstrated fast and effective route discovery across APIs, which makes it especially useful for APIs with simpler or more obvious structures. Despite being slower at times, In-Context was thorough in identifying every path. Because Chain-of-Thought takes a methodical approach, it can be useful in situations that call for in-depth, sequential analysis. This emphasizes how crucial it is to match the strategy to the features of the API in order to achieve the best results.

In conclusion, the results demonstrate that HackingBuddyGPT excels in route discovery, consistently outperforming RESTSpecIT across various API complexities. However, in some situations, RESTSpecIT works better for parameter discovery, which probably because it refines parameters iteratively. Although HackingBuddyGPT uses dynamically generated hints and API descriptions to its advantage, its approach might need to be improved further to better handle complex and less structured APIs. These results demonstrate the advantages of each framework and indicate that in order to attain more balanced performance across various API tasks, future iterations of HackingBuddyGPT should concentrate on improving parameter inference.

4.2.2 OpenAI o1

In this section, we explore the results of experiments using the o1 model. Table 4.9 provides a comparative analysis between our HackingBuddyGPT framework and the well-established RESTSpecIT. The table compares both frameworks on the basis of their ability to identify API routes and parameters across a variety of APIs. The results are categorized into two main metrics: the percentage of routes found and the percentage of parameters found, which allows for an easy side-by-side comparison of the performance of the framework.

API name	**% Found Routes**		**% Found Parameters**	
	HackingBuddyGPT	RESTSpecIT	HackingBuddyGPT	RESTSpecIT
An API of Ice and Fire	100%	98.57%	100%	93.53%
BALLDONTLIE NBA	100%	96.25%	71.43%	90.83%
Random User Generator	100%	100.00%	100%	73.33%
CoinCap	100%	77.00%	71.43%	80.66%
Open Brewery DB	100%	61.67%	100%	79.23%
ReqRes	100%	87.50%	42.86%	100.00%

Table 4.9: Comparison of best HackingBuddyGPT and RESTSpecIT results based with o1 on APIs and evaluation metrics

Table 4.9 compares the performance of HackingBuddyGPT and RESTSpecIT in identifying API endpoints and query parameters with o1 as the LLM. HackingBuddyGPT usually performs better than RESTSpecIT in discovering API endpoints across different API complexities. For simpler APIs like *An API of Ice and Fire*, both frameworks achieve almost success rates, with HackingBuddyGPT also identifying 100% of the query parameters. This high accuracy could be due to the straightforward nature of the API, which requires fewer complex routes and parameters to be discovered.

Route discovery For **simpler** APIs, HackingBuddyGPT performs well in route discovery since it identifies 100% of routes for *An API of Ice and Fire.* This suggests that HackingBuddyGPT works best when APIs have a clear and simple structure. Likewise, HackingBuddyGPT accurately detects all routes for the *Random User Generator API.*

HackingBuddyGPT yields good results in route discovery for more complex APIs because it finds all available routes (100%) for BALLDONTLIE NBA, matching RESTSpecIT. Additionally, it once more reaches 100% route discovery for CoinCap. All approaches perform similarly in Open Brewery DB, with HackingBuddyGPT achieving 100% route discovery and detecting no extra routes, which is probably due to the simple and predictable structure of the API.

For the most challenging API is ReqRes, HackingBuddyGPT finds 100% route discovery, while RESTSpecIT only achieves 95.24%.

Parameter discovery Although HackingBuddyGPT performs well at route discovery, the complexity of the API affects how well it performs in parameter discovery. Due to the straightforward design of the API, *An API of Ice and Fire* is able to detect all parameters with 100% accuracy. However, HackingBuddyGPT does not perform well on complex REST APIs. RESTSpecIT can detect 90.83% of the parameters for *BALLDONTLIE NBA*, while it can only detect 71.43%. The mutation-based refinement approach used in parameter discovery by RESTSpecITs is probably to blame for this.

HackingBuddyGPT finds every query parameter in *Random User Generator* with 100% accuracy. However, it only discovers 71.43% of parameters for the hard REST API *CoinCap* using , while RESTSpecIT manages to achieve 77.00%. However, for *Open Brewery DB* HackingBuddyGPT manages to find all parameters, which can be due to the simple nature of the parameters.

The ReqRes API is the hardest REST API in terms of parameter discovery for Hacking-BuddGPT. RESTSpecIT detects 100% of parameters, while HackingBuddyGPT only detects 42.86%. This illustrates the need to improve iterative refinement and parameter extraction, especially for complex APIs with dynamic structures.

In summary, HackingBuddyGPT is very good at finding API routes, especially in more complex cases. In contrast, RESTSpecIT is better at discovering parameters, particularly when they need to be fine-tuned step by step. Overall, both tools work well with simple APIs, but HackingBuddyGPT's approach for finding endpoints seems stronger in handling more complex REST API designs. Conversely, RESTSpecIT's parameter refinement technique excels in situations where accurate parameter identification is crucial.

Table 4.10 presents the evaluation metrics for HackingBuddyGPT across various APIs with o1 as the LLM. The table highlights HackingBuddyGPT's performance in discovering API endpoints, query parameters, and additional routes over 10 runs.

Effect of Prompt Strategies For *An API of Ice and Fire*, HackingBuddyGPT reaches 100% accuracy in route and parameter discovery for all prompt strategies (CoT, ToT,

API	% Found Routes			% Found Parameters			# Additional Found Routes		
	COT	TOT	ICL	COT	TOT	ICL	COT	TOT	ICL
An API of Ice and Fire	100%	100%	100%	100%	100%	100%	0	0	0
BALLDONTLIE NBA	100%	75%	75%	71.43%	25%	37.5%	0	0	0
Random User Generator	100%	100%	100%	100%	100%	100%	2	2	2
CoinCap	100%	100%	90.00%	71.43%	71.43%	71.43%	1	2	1
Open Brewery DB	100%	100%	100%	100%	100%	100%	0	0	0
ReqRes	100%	95.24%	95.24%	42.86%	4.76%	23.81%	40	90	95

Table 4.10: Average Evaluation Metrics for HackingBuddyGPT Across APIs over 10 runs with o1

and ICL). No extra or missing routes were found, showing that both HackingBuddyGPT and other tools can work well with this REST API because it is clear and well-structured.

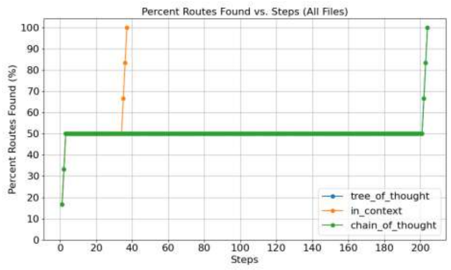
For *BALLDONTLIE NBA*, COT outperforms both TOT and ICL as it achieves 100% route discovery, while the other two reach only 75%. This could be because of the structured step-by-step reasoning of CoT, which likely helps COT to break down the API systematically. In contrast, TOT and ICL struggle to go beyond 75%, which could be because they either overcomplicate or fail to generalize the API's query patterns. In parameter discovery, COT again performs best at 71.43%, while TOT (25%) and ICL (37.5%) fall behind, which indicates that COT's logical breakdown better captures API input structures.

For the Random User Generator API, all three strategies perform very well, achieving 100% route and parameter discovery. They each detect two additional routes, which suggests that this API may have hidden or dynamically generated endpoints that the strategies interpret differently. The dynamic nature of this API allows all strategies to adapt well, leading to equal performance across the board.

For **hard** REST APIs like CoinCap, COT and TOT manage to achieve 100% route discovery, while ICL falls slightly behind at 90%. The API may need more effective prompt strategies or iterative refinement for extracting parameters, as all three struggle equally in parameter discovery (71.43%). Since ToT's branching logic investigates alternate paths more aggressively, even if they might not always be correct, COT finds one extra route while TOT detects two.

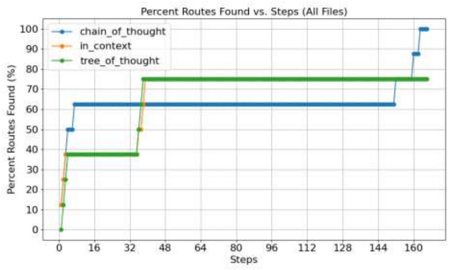
For Open Brewery DB, all strategies achieve 100% in both route and parameter discovery without detecting extra routes, reinforcing the idea that APIs with a simple and structured design are handled effectively regardless of the approach used.

When it comes to parameter discovery, ReqRes presents the largest challenge. TOT (4.76%) and ICL (23.81%) have much more difficulty than CoT, which only detects 42.86%. All strategies, nevertheless, are successful in route discovery. While TOT and ICL slightly decline to 95.24%, COT continues to lead in route discovery (100%). This suggests that a more structured reasoning approach, such as CoT, is necessary for complex APIs with highly structured or dynamic parameters. With COT detecting 40, TOT detecting 90, and ICL detecting 95, the number of additional routes found is also incredibly high. This implies that TOT and ICL might be over-examining routes, which could result in false positives during route discovery.

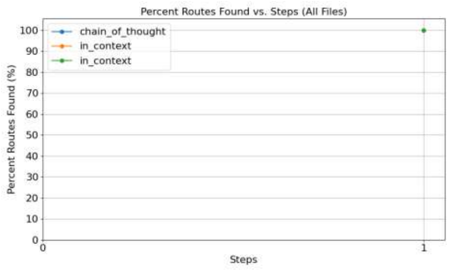


(a) An API of Fire and Ice

%



(b) BALLDONTLIE NBA



(c) Random User Generator

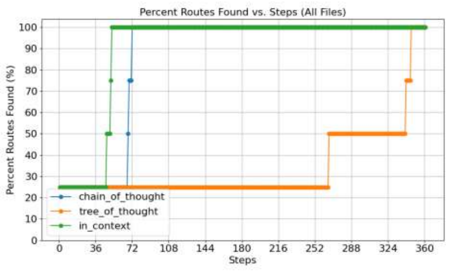
Figure 4.3: Number of steps in route discovery for o1 of simple and medium benchmarks

In summary, HackingBuddyGPT proves to be highly effective in discovering routes, particularly in simpler and moderately complex APIs. RESTSpecIT, on the other hand, demonstrates superior performance in parameter discovery, especially for APIs with more complex structures. Whileboth tools excel with simpler APIs, HackingBuddyGPT is more robust in handling complex endpoint structures, while RESTSpecIT's strength lies in the iterative refinement of parameters. Both tools also show the ability to identify additional routes, with HackingBuddyGPT generally outperforming in this area.

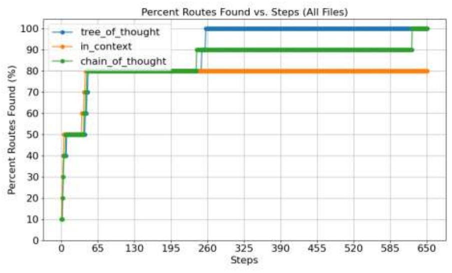
Figure 4.3 illustrates the performance of various route discovery strategies applied tosix different APIs using the o1 model. InAPI (a) *Fire and Ice*, both the Tree-of-Thought and Chain-of-Thought methods quickly reach a plateau in route discovery, typically after a few 40 steps. In contrast, the In-Context Learning method discoveredall endpoints at around 40steps. Compared to Figure 4.1 (a), it is evident that o1 takes considerably longerto discover endpoints than GPT-4o-mini, highlighting a key performance difference between the two models.

In (b) *BALLDONTLIE NBA*, Chain-of-Thought excels in uncovering routes rapidly, outperforming Tree-of-Thought, which takes a longer time to reach full route discovery. In contrast, In-Context Learning andTree-of-Thoughtonlymanagetofindaround75%,however,thistheyachievedtod
of-Thought,Interestingly,forthisRESTAPI,theperformanceofGPT-4o-minicloselymatchesthatofo1,showing

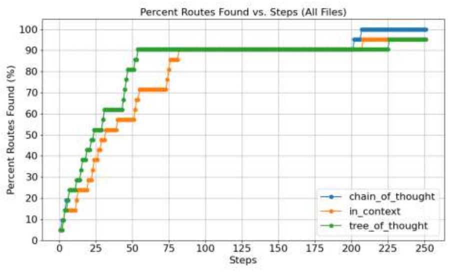
strategies achieved to find the endpoint very fast. In particular,o1discoverstheoneendpointforallpromptstrategiesri
4o-minitakesslightlylongertofindthesameresult,furthersuggestingthato1mayhavecertainadvantagesinspecifictypeso



(a) OpenBreweryDB



(b) CoinCap



(c) Reqres

Figure 4.4: Number of steps in route discovery for o1 of hard benchmarks

Similarly, in Figure 4.4 (a) *Open Brewery DB*, Chain-of-Thought and In-Context Learning quickly identify most of the routes, while Tree-of-Thought lags behind. In this case, GPT-4o-mini is much faster than o1 when using In-Context Learning, needing only about 41 steps to find all endpoints, while o1 takes around 360 steps. With Chain-of-Thought, both models reach 100% after about 50 steps, though o1 is slightly faster, finishing in 45 steps. Tree-of-Thought also works a bit better in o1, reaching full discovery in 50 steps compared to 55 steps in GPT-4o-mini.

In (b) *CoinCap*, all methods demonstrate slower route discovery, with Chain-of-Thought leading and Tree-of-Thought trailing the others. A similar trend is observed when comparing endpoint discovery between o1 and GPT-4o-mini: o1 requires about 650 steps to reach full endpoint discovery, while GPT-4o-mini only needs around 100 steps.

Finally, in (c) *Reqres*, the trends mirror those seen in previous APIs, with Chain-of-Thought reaching high route discovery early, Tree-of-Thought progressing slowly, and In-Context Learning gradually increasing to a later plateau. Interestingly, o1 can discover 100% of the routes faster than GPT-4o-mini in this case, providing a surprising result that contrasts with the findings for other APIs.

Overall, these results show a clear pattern: GPT-4o-mini is consistently faster in discovering endpoints than o1, suggesting that GPT-4o-mini may be better suited for endpoint discovery tasks. This observation highlights differences in the efficiency of these models for specific applications, with o1 performing more slowly across most APIs in comparison.

4.3 Results: WebAPITesting

4.3.1 OpenAI GPT-4o-mini

HackingBuddyGPT was tested on different benchmarks to find various API vulnerabilities. The tables below show the results, including which vulnerabilities were found and how they were detected.

Benchmark	% Vulns Found	Vuln Types	Vuln Endpoints
TicketBuddy	100%	6	4
OWASP vAPI	100%	6	14
OWASP crAPI	75%	4	22
OWASP Juice Shop	60%	5	-

Table 4.11: Best vulnerability detection results of our approach

Table 4.11 provides an overview of the percentage of vulnerabilities found and the total amount of vulnerabilities detected across different benchmarks. As the table shows, HackingBuddyGPT did a great job detecting vulnerabilities, hitting 100% on TicketBuddy (6 types) and OWASP vAPI (5 types). It struggled a bit more with tougher benchmarks like OWASP crAPI and Juice Shop, where it caught 75% and 60% of the issues. This may reflect the added complexity or depth of those APIs.

Benchmark	Vulnerability	How Vulnerability was Found	Detected ?
TicketBuddy	Broken Access Control	Tried to make an API request to access other users' data.	✓
TicketBuddy	Broken Function Level Authorization	Tried to access admin-only functions to check if proper Authorization is enforced.	✓
TicketBuddy	Security Misconfiguration	Reviewed the API's security configuration for common misconfigurations, such as improper HTTP headers, exposed services, verbose error messages, or default credentials.	✓
TicketBuddy	Identification and Authentication Failures	Tested the Authentication mechanism by attempting brute-force or credential stuffing attacks using weak, default, or leaked passwords.	✓
TicketBuddy	Lack of Resources and Rate Limiting	Performed a DoS attack or sent excessive requests to test if the API limits or blocks excessive calls. A lack of rate-limiting can lead to brute-force or DoS vulnerabilities.	✓
TicketBuddy	Injection (SQL, etc.)	Tried injecting malicious input (e.g., SQL, NoSQL) into API parameters or request bodies.	✓

Table 4.12: Vulnerabilities Detected via HackingBuddyGPT for TicketBuddy

Table 4.12 outlines the specific vulnerabilities found in *TicketBuddy* via HackingBuddyGPT. These vulnerabilities include a variety of security issues such as Broken Access

Control , Broken Function Level Authorization, Security Misconfiguration, Identification and Authentication Failures, and Injection vulnerabilities.

Broken Access Control It was detected by attempting to make an API request to access other users' data, which is a common vulnerability in poorly configured Authorization systems. HackingBuddyGPT was able to detect unauthorized access when a user with ID 45 tried to access data belonging to user ID 44.

Broken Function Level Authorization This vulnerability was found by trying to access admin-only functions, ensuring that proper Authorization was enforced before allowing access. HackingBuddyGPT found this vulnerability by trying to access endpoints that should be restricted.

Security Misconfiguration It was found by checking the API's security settings for common problems like missing HTTP headers, exposed services, detailed error messages, or default passwords. While HackingBuddyGPT was able to detect this in some cases, it was not always successful in identifying all security misconfigurations.

Identification and Authentication Failures This vulnerability was discovered by testing the Authentication mechanism with brute-force or credential stuffing attacks using weak, default, or leaked passwords. This was successfully detected by HackingBuddyGPT, highlighting the importance of strong Authentication measures.

Injection (SQL, etc.) It was tested by trying to send harmful input (like SQL or NoSQL injection) through API parameters or request bodies. HackingBuddyGPT was able to find these injection flaws, which are important to fix to avoid data leaks or unauthorized access.

However, HackingBuddyGPT did not always catch certain vulnerabilities. For example, the **Lack of Resources** and **Rate Limiting** issue was tested by sending too many requests to check if the API would block or limit them. This problem was not always found, which shows that HackingBuddyGPT may sometimes miss **Rate-Limiting** issues. These are important to catch because they can lead to DoS or brute-force attacks. Moreover, **Security Misconfiguration** was not always found, which is possibly due to the subtle nature of certain misconfigurations or how the API responds under specific conditions.

To better understand how well HackingBuddyGPT finds vulnerabilities, a confusion matrix was created to show its results on the **TicketBuddy** API.

Confusion matrix Figure 4.5 shows perfect results because the model found all 4 vulnerable endpoints (TP) with no mistakes. It did not find any missed cases (FN = 0) and no wrong detections (FP = 0). However, it did not find any safe endpoints, so there are no true negatives (TN).

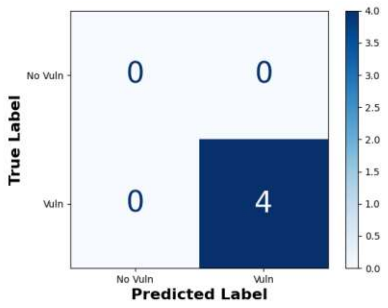


Figure 4.5: Confusion matrix of TicketBuddy

Overall, HackingBuddyGPT proved effective at spotting key vulnerabilities in TicketBuddy, especially issues with Authentication, Authorization, and Injection Attacks. However, it still struggled with more complex problems like Rate-Limiting and hidden misconfigurations, which shows that there is room for improvement.

Benchmark	Vulnerability	How Vulnerability was Found	Detected?
vAPI	Identification and Authentication Failures	Tested the Authentication mechanism by attempting brute-force or credential stuffing attacks using weak, default, or leaked passwords.	✓
vAPI	Injection (SQL, etc.)	Tried injecting malicious input (e.g., SQL, NoSQL) into API parameters or request bodies.	✓
vAPI	Broken Function Level Authorization	Tried accessing higher-privileged API functions by modifying the endpoint or parameters (e.g., accessing admin-only functions) to check if proper Authorization is enforced.	✓
vAPI	Security Misconfiguration	Reviewed the API's security configuration for common misconfigurations, such as improper HTTP headers, exposed services, verbose error messages, or default credentials.	✓
crAPI	Excessive Data Exposure	Inspected responses from the "Community" page for sensitive information like another user's details exposed in API responses.	✗
crAPI	SQL Injection	Injected SQL queries (e.g., '0' OR '0' = '0') into the coupon redemption process to manipulate the database and redeem coupons.	✓
crAPI	Unauthenticated Access	Sent an unauthenticated request to an endpoint that provides data (e.g., order details) without requiring Authentication.	✓
crAPI	Mass Assignment	Manipulated input fields by sending extra fields or modifying object properties through API calls (e.g., updating user roles).	✗

Table 4.13: Vulnerabilities Detected via HackingBuddyGPT for vAPI and crAPI

Table 4.13 shows the vulnerabilities that HackingBuddyGPT detected in two APIs: vAPI and crAPI. It includes what issues were found, how they were tested, and whether HackingBuddyGPT was able to catch them.

In the case of vAPI, HackingBuddyGPT successfully identified all vulnerabilities that can be detected through REST API calls alone.

Identification and Authentication Failures This issue was found by checking how the REST API handles login attempts. HackingBuddyGPT simulated attacks using weak or default passwords, similar to a credential stuffing scenario, to see if it could break through the authentication system. HackingBuddyGPT identified that the system was vulnerable to such attacks, emphasizing the importance of strong password policies and rate-limiting mechanisms for Authentication processes.

Injection (SQL, etc.) It was tested via Injection attacks, particularly SQL and NoSQL Injections. This was done by injecting malicious input into API parameters or request bodies. These types of attacks try to trick the database into doing something it should not do like revealing private data or bypassing login checks. HackingBuddyGPT was able to detect these issues, showing how important it is to properly check and sanitize inputs to stop unauthorized access to databases or systems.

Broken Function Level Authorization This vulnerability was identified by attempting to access higher-privileged functions in the API by modifying the endpoint or API parameters. For example, the attempt to access admin-only functions without proper Authorization checks was tested. HackingBuddyGPT successfully detected this flaw, highlighting the need for robust Authorization checks at every function level to make sure that only users, which are authorized, have access the functionality,.

Security Misconfiguration HackingBuddyGPT also found signs of misconfiguration, like missing security headers, exposed services, or error messages giving too much detail. Problems like default credentials and weak settings can leave APIs wide open to attackers. HackingBuddyGPT detected multiple instances of security misconfigurations, highlighting the importance of a secure by design configuration and regular audits of API settings to prevent exploitation. To test how well HackingBuddyGPT detects vulnerabilities, we analyzed its results on the vAPI API using a confusion matrix (Figure 4.6).

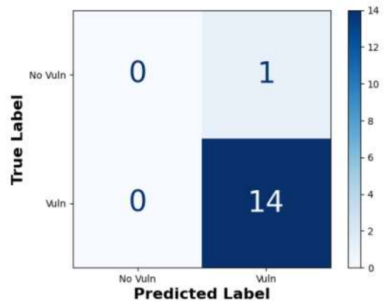


Figure 4.6: Confusion matrix of vAPI

Confusion matrix The model found all 14 real vulnerabilities (TP) with no false negatives, giving it a perfect recall score. While HackingBuddyGPT performed well in detecting vulnerabilities in *vAPI*, it also flagged five safe endpoints as vulnerable. That means it had no true negatives in this case and showed a slight tendency to overpredict. However, it reached a high precision and recall (93.33%), with a strong F1-score showing overall balance between detection and reliability.

Overall, HackingBuddyGPT did a good job finding all the real vulnerabilities in *vAPI*, with high recall and precision. However, it flagged some safe endpoints by mistake, which shows it might overpredict problems. The results show that the tool is strong at spotting real threats, but could be improved to reduce false alarms.

For *crAPI*, the results were more mixed. It consistently identified 2 out of 4 vulnerabilities and occasionally caught a third.

Excessive Data Exposure happens when an API response shows sensitive information, like data from another user. In crAPI, the "Community" page was checked for this issue, as it showed information that should have been hidden. However, HackingBuddyGPT did not catch this problem, which suggests the tool may struggle to detect data exposure when the information is hidden in complex responses or not properly filtered on the client side. It shows a weakness in finding this type of issue in certain situations.

SQL Injection was attempted on the coupon redemption process in *crAPI*. HackingBuddyGPT successfully detected the vulnerability by using the Injection of SQL queries like '0' OR '0' = '0', which could manipulate the underlying database. This emphasizes the critical need for input validation to prevent unauthorized database access, and HackingBuddyGPT effectively handled this detection.

Unauthenticated Access was successfully identified by HackingBuddyGPT when unauthenticated requests were sent to an endpoint that provided sensitive data, such as order details. The tool detected the issue, confirming the importance of proper Authentication mechanisms for sensitive data access in APIs.

Mass Assignment occurs when extra fields or modified object properties are sent in a REST API request, which can lead to unauthorized actions or privilege escalation. For example, changing user roles via API calls was tested and detected successfully by HackingBuddyGPT. This highlights the need for careful validation of input data to prevent unauthorized changes to the system.

A confusion matrix was used to further evaluate HackingBuddyGPT's classification results on the *crAPI* API in order to determine how well it detected vulnerabilities.

Confusion matrix *crAPI*'s confusion matrix (see 4.7) offers a thorough analysis of the model's classification performance. The actual vulnerabilities the model successfully identified are indicated by the 17 True Positives. Five vulnerabilities that were overlooked by the model are known as False Negatives (FN). Five False Positives (FP), or non-vulnerable scenarios that the model incorrectly identified as vulnerabilities, were found.

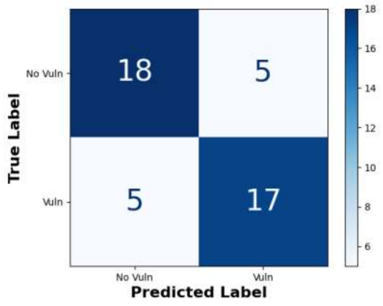


Figure 4.7: Confusion matrix crAPI

There were 18 True Negatives (TN) = 18, or non-vulnerabilities that the model accurately recognized.

The model does well in differentiating between vulnerable and non-vulnerable cases, with an accuracy of about 77.78%. However, accuracy by itself might not be enough to assess the model's dependability, particularly when there is a class imbalance. The bulk of the reported vulnerabilities are, in fact, correct, according to the precision score of 77.27%. The existence of false positives, however, raises the possibility that certain non-vulnerable circumstances are being mistakenly identified as vulnerabilities. The model's 77.27% recall score indicates that it can detect a sizable percentage of current vulnerabilities. The existence of false negatives, however, suggests that certain vulnerabilities are still undiscovered. The model's 77.27% value indicates that it strikes a decent balance between identifying vulnerabilities and preventing misclassification.

The results show that HackingBuddyGPT was able to find most of the vulnerabilities in *crAPI*, including important ones like SQL Injection. However, it missed the Excessive Data Exposure vulnerability in some cases. This suggests it may have trouble spotting this type of problem, especially when the exposed data isn't easy to see or is hidden by how the API is set up. Still, HackingBuddyGPT proved useful for automating security testing, catching common issues like unauthenticated access and helping highlight areas where it can be improved.

Table 4.14 provides a detailed look at vulnerabilities detected in *OWASP Juice Shop*. HackingBuddyGPT was able to identify common vulnerabilities such as Identification and Authentication Failures, NoSQL Injection, and Information Disclosure via Headers. However, it struggled to consistently identify Sensitive Data Exposure and Rate Limiting issues. These limitations indicate that while HackingBuddyGPT performs well in detecting many vulnerabilities, there are some complex cases, particularly related to Rate Limiting and data exposure, that it may not always catch.

Identification and Authentication Failures It was discovered in the case of the *OWASP Juice Shop*, by trying out week passwords via brute-force attempts. HackingBud-

Benchmark	Vulnerability	How to Find the Vulnerability	Detected?
OWASP Juice Shop	Identification and Authentication Failures	Test the login and password reset endpoints with weak or brute-force attempts to exploit Authentication mechanisms.	✓
OWASP Juice Shop	Sensitive Data Exposure	Inspect API responses for unencrypted sensitive data such as credit card numbers or personal information.	✗
OWASP Juice Shop	Broken Access Control	Modify parameters like user IDs in API requests (e.g., change '/rest/basket/1' to '/rest/basket/2') to access unauthorized data.	✓
OWASP Juice Shop	Insufficient Rate Limiting	Make repeated requests to endpoints (e.g., login, registration) and check if the system limits the number of attempts.	✗
OWASP Juice Shop	Mass Assignment	Manipulated input fields by sending extra fields or modifying object properties through API calls (e.g., updating user roles).	✗
OWASP Juice Shop	NoSQL Injection	Injected NoSQL payloads (e.g., " or 1=1 —' into email field) in the API request to manipulate database queries.	✓
OWASP Juice Shop	Information Disclosure via Headers	Inspected HTTP headers from API responses for sensitive data such as server details or API version information.	✓

Table 4.14: Vulnerabilities Detected via HackingBuddyGPT for OWASP Juice Shop

dyGPT successfully identified this vulnerability through REST API testing, highlighting the importance of strong Authentication mechanisms.

Sensitive Data Exposure This vulnerability can lead to the exposure of sensitive information, such as credit card details or personal data. The API responses were examined for unencrypted data leaks. However, this vulnerability was not always detected by HackingBuddyGPT, indicating potential limitations in identifying subtle data exposure issues that depend on specific API response formats or encryption practices.

Broken Access Control It was tested by modifying request parameters (e.g., changing user IDs in API calls like /rest/basket/1 to /rest/basket/2) to access unauthorized resources. HackingBuddyGPT effectively detected unauthorized access through REST API testing, emphasizing the need for strict access control measures in API implementations.

Insufficient Rate Limiting This vulnerability was assessed by making multiple rapid requests to endpoints such as log-in or registration. This vulnerability, which can allow brute-force attacks if not properly mitigated, was not always detected by HackingBuddyGPT. This suggests that some rate-limiting mechanisms might be challenging to identify without simulating real-world attack patterns over time.

Mass Assignment It happens when attackers modify object properties by adding or changing fields in API requests, such as unauthorized user role changes. The tests on *OWASP Juice Shop* showed that HackingBuddyGPT did not always catch this issue, suggesting it may miss unauthorized changes through API calls.

NoSQL Injection This security vulnerability involves injecting malicious NoSQL queries (e.g., using payloads like ' or 1=1 —) into API requests to manipulate database queries.

In this case, HackingBuddyGPT successfully detected NoSQL Injection vulnerabilities, demonstrating its effectiveness in identifying Injection flaws in REST APIs.

Information Disclosure via Headers It was evaluated by inspecting HTTP headers for sensitive information, such as server details or API version numbers. HackingBuddyGPT detected this vulnerability effectively, highlighting the need to sanitize or remove unnecessary information from HTTP headers in API responses.

Confusion matrix Since the true number of non-vulnerable endpoints for OWASP Juice Shop was not provided, we could not create a confusion matrix. A confusion matrix requires four key values: true positives (correctly identified vulnerabilities),true negatives (correctly identified non-vulnerabilities), false positives (incorrectly identified vulnerabilities), and false negatives (missed vulnerabilities). Without knowing the false positives, it is impossible to measure the model's precision, recall, or overall performance. Our assessment revealed that 60% of the vulnerabilities we identified were discovered.

Overall, these tables show that HackingBuddyGPT is good at finding common API vulnerabilities across different tests. While it works well in most cases, it sometimes misses certain issues, especially with rate limiting, data exposure, and access control flaws. This suggests that while HackingBuddyGPT is powerful, it could be improved to better detect complex security issues. Still, the tool did well at spotting important vulnerabilities like Authentication Failures, Injection Attacks, and Broken Authorization, which are serious API security concerns. HackingBuddyGPT shows it can automate security testing by finding common API problems, saving time and effort. While it sometimes missed issues like Rate Limiting, Data Exposure, and access control flaws, it generally does a good job detecting common vulnerabilities. This suggests HackingBuddyGPT could be improved to spot more complex issues.

4.3.2 Summary

In summary, these experiments show how well the two LLMs (GPT-4o-mini and o1,) generate test cases for REST APIs and find security vulnerabilities. Overall, GPT-4o-mini was especially good at finding endpoints, even in complex APIs, and creating test cases using structured prompts like Chain-of-Thought. It also did better than RESTSpecIT at finding routes, but struggled with discovering parameters in more complex cases. In comparison, o1 did not perform that well. HackingBuddyGPT, which uses GPT-4o-mini, performed well at spotting common issues like authentication failures and injection attacks, but had difficulty with more complex problems like rate limiting and data exposure. Overall, the study highlights the strengths of these tools but also shows they need improvement. With better training and better prompt strategies, they could become even more effective at catching harder-to-find security flaws in real-world APIs.

CHAPTER 5

Discussion

This discussion answers the initial research questions by analyzing the results of Chapter 4.

Research Question 1: *What is the efficacy of various Large Language Models (LLMs) in generating test cases for REST APIs, and which factors significantly influence their performance?*

The WebAPIDocumentation experiments show that how well Large Language Models (LLMs) like GPT-4o-mini and o1 can generate test cases for REST APIs depends on the complexity of the API, the model's abilities, and the prompts used. This is shown by comparing HackingBuddyGPT and RESTSpecIT. Similar to RESTSpecIT, HackingBuddyGPT performs black-box testing by automating the discovery of routes and parameters through schema analysis. These experiments test how well the models can identify endpoints, parameters, and edge cases, which are key for creating test cases. Input validation is also tested during the discovery process, as if a 401 or 422 error occurs, the endpoint is queried again with different parameters.

The results show that HackingBuddyGPT is great at finding routes, especially in complex APIs, but it struggles with discovering parameters, particularly for APIs that need repeated adjustments. This highlights a challenge in creating test cases, where models must generate valid cases with the right routes and parameters. While HackingBuddyGPT is good at finding endpoints, it has trouble with dynamic parameters that need context.

The experiments also show that API complexity greatly affects how well models perform. For simple APIs, generating test cases is easy, and models quickly find routes and parameters. But as APIs get more complex, creating accurate test cases becomes harder. The performance depends on three key factors: API complexity, model abilities, and how the prompts are designed.GPT-4o-mini seems to have a stronger understanding of RESTful API structures, allowing it to find hidden or undocumented endpoints better than o1. Prompt Engineering helps HackingBuddyGPT use context and structured input to create more accurate test cases. Its advanced reasoning lets it figure out missing endpoints, which helps with complex or dynamic APIs. It also generalizes better, using knowledge from one API for others, while o1 is more limited.Strategies such as Chain of Thought (COT) and Tree of Thought (TOT) improve the accuracy of the test case by guiding the model through a structured reasoning process, whereas ICL does not achieve the same results. These advantages make GPT-4o-mini more efficient in generating test cases for a wide variety of APIs than o1.

In conclusion, as previously stated, Prompt Engineering, model capabilities, and API complexity all affect how well LLMs generate REST API test cases. Due to training and design differences, GPT-4o-mini performs better than o1 in identifying the REST API endpoint and parameters. GPT-4o-mini's cost-effectiveness and processing efficiency make it ideal for tasks requiring rapid route and parameter identification. In contrast, o1 is designed with a focus on complex reasoning and problem solving, which, while enhancing its performance in tasks like coding and mathematics, may result in slower response times and higher costs. These distinctions suggest that GPT-4o-mini's training emphasizes rapid and efficient comprehension of the API structure, whereas o1 prioritizes deep reasoning capabilities [AI24, VIS24].

Research Question 2: *What Prompt Engineering techniques are most effective for generating test cases for REST APIs using LLMs?*

The WebAPIDocumentation experiments show that Chain of Thought (COT) and Tree of Thought (TOT) are the best strategies for generating test cases for REST APIs. COT performs especially well at finding parameters and endpoints, especially in complex APIs like CoinCap and BALLDONTLIE NBA. This is likely because COT breaks tasks into smaller steps, helping the model find missing parameters more accurately. Previous research has shown that COT improves logical reasoning by breaking down complex tasks into structured steps, which helps with accuracy in tasks that need detailed inference [WWS+22]. It seems that clear and specific instructions helped the model identify key components of the API, reducing redundancy, and ensuring relevant, contextually appropriate test cases [**?**]. TOT also improves API exploration by systematically breaking down discovery tasks, making it particularly effective in structured API environments [**?**].

In contrast, In-context Learning (ICL) performs the worst in finding parameters, which is most likely because it depends on the provided context, which may not always have enough information to spot hidden parameters. While ICL works well for tasks with clear context, its ability to find API parameters is limited. However, it can still be effective for simple APIs, like the Random User Generator API, where it achieved 100For finding routes, all three strategies do well, but COT and TOT are usually better at discovering hidden endpoints, especially in complex APIs. This matches previous research, which shows that structured reasoning methods help improve performance in data exploration tasks. The results also show that GPT-4o-mini consistently outperforms o1 in finding API endpoints, suggesting that differences in training and model optimization for structured

API understanding affect performance in API testing. In conclusion, COT is the best strategy for finding parameters and endpoints. TOT is performed also well at finding endpoints. Future work should focus on improving these strategies to handle complex parameter structures better.

Research Question 3: *To what extent can LLMs identify potential failure points or vulnerabilities within REST APIs, and how accurate are their predictions?* The findings demonstrate that HackingBuddyGPT can identify REST API vulnerabilities, though its effectiveness varies according to how difficult the benchmarks are. In the WebAPITesting experiments, HackingBuddyGPT performed well in detecting vulnerabilities in simpler benchmarks like TicketBuddy and OWASP vAPI, achieving 100

HackingBuddyGPT performed well on simpler benchmarks but was less reliable with more complex ones, like OWASP crAPI and OWASP Juice Shop. While it found many serious vulnerabilities like SQL injection and unauthenticated access, it missed issues like excessive data exposure and sensitive data exposure, which are harder to detect. Moreover, it excels at identifying common vulnerabilities but falters at identifying subtler or more complicated problems that require more thorough investigation. In OWASP Juice Shop, for instance, the tool detected issues like NoSQL Injection and Broken Authentication, but it struggled with issues like mass assignment and inadequate rate limiting, which are more difficult to test with simple API calls. This demonstrates the tool's inability to identify more complicated vulnerabilities that require a more thorough examination of the API's operation in various scenarios. In conclusion, HackingBuddyGPT is good at finding serious defects, but not perfect and sometimes misses vulnerabilities. The tool needs to be improved in these areas, especially when handling more complex cases.

5.1 Challenges

While the LLMs worked well overall, we ran into a few challenges while building the two use cases. One big challenges was stopping the model from repeating the same queries because it kept using endpoints that it had already found. We solved this by keeping track of both successful and failed requests so the model could explore more without getting stuck. Figuring out the right parameters for each endpoint was also tricky. Even with example queries, the model had to create new versions for each request. Therefore, we had to be careful, as one wrong parameter could mess up the test.

Another issue was with In-Context Learning (ICL) because it sometimes copied query patterns too exactly from the examples instead of adjusting them to fit the current situation. This means that it executed requests on the provided examples. In order to fix this issue we adjusted the prompt. Finally, it was challenging to get consistent results across all benchmarks, since each one had its own structure, endpoint logic, and security setup. This called for a versatile framework that could handle a wide range of API conventions. It was difficult but necessary to generalize test cases for these diverse structures, requiring specialized approaches to parameter generation and endpoint discovery.

Another major challenge was developing a standardized testing approach that could accommodate different API configurations. To conduct effective security tests, the framework needed to be universally applicable across benchmarks while remaining adaptable to each API's unique characteristics.

The generation of Python test cases also presented a challenge. Although the tests were initially generated, they often contained placeholders or lacked full functionality. Therefore, it was necessary to manually restructure and modify the generated tests so that they could be run in actual situations. The process of improving the model's code generation capabilities was essential because it made clear that, especially for more complicated test cases, the model must generate fully functional executable code.

These difficulties show the need for a testing framework that is more resilient and flexible. A framework like this needs to be able to handle a variety of API structures and guarantee that the code produced can be used directly in security testing without the need for extensive post-processing or manual modifications.

CHAPTER 6

Conclusion

This thesis aimed to evaluate the efficacy of Large Language Models (LLMs) in generating test cases for REST APIs, identify effective Prompt Engineering techniques, and assess their ability to detect vulnerabilities within REST APIs. Two new use cases were added to HackingBuddyGPT (WebAPIDocumentation and WebAPITesting) to answer these questions. WebAPIDocumentation generates OpenAPI specifications (and adressed the problem of inconsitent Openapi specifications) in a black-box setting and in this process conducts fuzzing. WebAPITesting focuses on more advanced testing, such as determining security vulnerabilities such as Broken Authentication.

The findings of the experiments of WebAPIDocumentation show that for GPT-4o-mini is better suited for endpoint and parameter discovery than o1. GPT-4o-mini seems to have a better understanding of API structures, allowing it to identify hidden endpoints and generate accurate test cases through improved Prompt Engineering. The prompting strategy that performed the best as COT can be attributed to the fact that these kinds of prompts are structured. Additionally, adjusting In-Context Learning (ICL) to encourage creativity and varying query paths helped the model avoid repetition and explore a wider range of possible scenarios.

The results of WebAPITesting show that GPT-4o-mini was successful in identifying common security vulnerabilities such as broken authentication and NoSQL injection, but struggled with more complex issues like Sensitive Data Exposure and Insufficient Rate Limiting. While its accuracy was higher with well-documented APIs, it showed variability in performance, suggesting that further refinement in training and Prompt Engineering is needed to improve detection of complex vulnerabilities. Using structured prompts, example queries, and providing context about the API helped the model better identify the right endpoints and create more accurate test cases.

In conclusion, this thesis contributes to the current field by showing that LLMs can be used in generating test cases for REST APIs and are efficient in doing so. Their performance can be improved by using Prompt Engineering strategies such as COT. Moreover, it showed that LLMs can be used to uncover common security vulnerabilities.

6.1 Lessons Learned

The study provided several valuable lessons. First, improved Prompt Engineering played an importnat role in enhancing the performance of LLMs, guiding them to generate more accurate and contextually relevant test cases. Second, while GPT-4o-mini excelled in detecting common vulnerabilities, it showed limitations in identifying more complex security issues, indicating the need for further refinement. Finally, the importance of contextual awareness became clear: providing the model with clear information about the API structure in the form of a complete OpenAPI specification significantly improved both the accuracy of its predictions and the quality of the generated test cases.

6.2 Future Work

Future work in this area could focus on improving the detection of complex vulnerabilities, such as sensitive data exposure and rate limiting, by incorporating more training in challenging scenarios. Increasing the diversity of training datasets to include various API architectures and security vulnerabilities would enhance the model's generalization capabilities across distinct API categories. Additionally, more research into dynamic Prompt Engineering, where the model tailors prompts based on the specifics of the API, could lead to more efficient and accurate test case generation. Future iterations could also focus on automating the suggestion of fixes or mitigation strategies, making LLMs tools not only for vulnerability detection but also for remediation.

Overview of Generative AI Tools Used

In this thesis, two AI models, ChatGPT-4o-mini and ChatGPT-4o1, were used to improve task efficiency and accuracy. They helped in conducting experiments, generating code snippets, suggesting test scenarios, and automating tasks. GPT-4 specifically helped format tables and refine text for improved clarity and coherence. AI tools were used solely for technical support and did not influence result interpretation. All AI-generated content was reviewed for academic integrity.

Übersicht verwendeter Hilfsmittel

In dieser Arbeit wurden zwei KI-Modelle, ChatGPT-4o-mini und ChatGPT-4o1, verwendet, um die Aufgabeneffizienz und -genauigkeit zu verbessern. Sie halfen bei der Durchführung von Experimenten, der Generierung von Codeausschnitten, der Anregung von Testszenarien und der Automatisierung von Aufgaben. GPT-4 hat insbesondere dabei geholfen, Tabellen zu formatieren und verfeinern des Texts, um die Klarheit und Verständlichkeit zu verbessern. KI-Tools dienten ausschließlich der technischen Unterstützung und hatten keinen Einfluss auf die Ergebnisinterpretation. Alle KI-generierten Inhalte wurden auf akademische Integrität überprüft.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABC$^+$21] Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Benjamin Mann, Nova DasSarma, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Jackson Kernion, Kamal Ndousse, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, and Jared Kaplan. A general language assistant as a laboratory for alignment. *CoRR*, abs/2112.00861, 2021. URL: `https://arxiv.org/abs/2112.00861`, `arXiv:2112.00861`.

[Agg23] Adriti Aggarwal. 138 question on number, ranking and time sequence?, Jun 2023. Accessed: 2025-04-02. URL: `https://morelifechanger.in/138-question-on-number-ranking-and-time-sequence/`.

[AGP19] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: stateful REST API fuzzing. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 748–758. IEEE / ACM, 2019. `doi:10.1109/ICSE.2019.00083`.

[AI24] Vellum AI. Analysis: Openai o1 vs gpt-4o, 2024. Accessed: 2025-04-01. URL: `https://www.vellum.ai/blog/analysis-openai-o1-vs-gpt-4o`.

[AM23] Sattam J. Alharbi and Tarek Moulahi. Api security testing: The challenges of security testing for restful apis. *International Journal of Innovative Science and Research Technology (IJISRT)*, 8(5):1485–1499, 2023. Accessed: 2025-04-01. `doi:10.5281/zenodo.7988410`.

[AMS$^+$23] Juan C. Alonso, Alberto Martin-Lopez, Sergio Segura, José María García, and Antonio Ruiz-Cortés. ARTE: automated generation of realistic test inputs for web apis. *IEEE Trans. Software Eng.*, 49(1):348–363, 2023. `doi:10.1109/TSE.2022.3150618`.

[Arc19a] Andrea Arcuri. Many independent objective (MIO) algorithm for test suite generation. *CoRR*, abs/1901.01541, 2019. URL: `http://arxiv.org/abs/1901.01541`, `arXiv:1901.01541`.

[Arc19b] Andrea Arcuri. Restful API automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1):3:1–3:37, 2019. doi:10.1145/3293455.

[Ber24] Dave Bergmann. What is a context window?, Nov 2024. Accessed: 2025-04-02. URL: https://www.ibm.com/think/topics/context-window.

[BFGC21] Ovidiu Banias, Diana Florea, Robert Gyalai, and Daniel-Ioan Curiac. Automated specification-based testing of REST apis. *Sensors*, 21(16):5375, 2021. URL: https://doi.org/10.3390/s21165375, doi:10.3390/S21165375.

[BGMS21] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In Madeleine Clare Elish, William Isaac, and Richard S. Zemel, editors, *FAccT '21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021*, pages 610–623. ACM, 2021. doi:10.1145/3442188.3445922.

[Bha24] Bhuwan Bhatt. Self-calibration prompting, Sept 2024. Accessed: 2025-04-02. URL: https://learnprompting.org/docs/advanced/self_criticism/self_calibration.

[BIEC20] Steven Bucaille, Javier Luis Cánovas Izquierdo, Hamza Ed-Douibi, and Jordi Cabot. An openapi-based testing framework to monitor non-functional properties of REST apis. In Mária Bieliková, Tommi Mikkonen, and Cesare Pautasso, editors, *Web Engineering - 20th International Conference, ICWE 2020, Helsinki, Finland, June 9-12, 2020, Proceedings*, volume 12128 of *Lecture Notes in Computer Science*, pages 533–537. Springer, 2020. doi:10.1007/978-3-030-50578-3_39.

[BJN$^+$22] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, Scott Johnston, Shauna Kravec, Liane Lovitt, Neel Nanda, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, Benjamin Mann, and Jared Kaplan. Training a helpful and harmless assistant with reinforcement learning from human feedback. *CoRR*, abs/2204.05862, 2022. URL: https://doi.org/10.48550/arXiv.2204.05862, arXiv:2204.05862, doi:10.48550/ARXIV.2204.05862.

[BMR$^+$20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish

Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: `https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`.

[BT24] Slaviša Avramović Baeldung Team. Intro to jacoco, January 2024. Accessed: 2025-04-01. URL: `https://www.baeldung.com/jacoco`.

[Cho21] Deepak Chourasia. robotframework-restlibrary 1.0, August 2021. Accessed: 2025-04-02. URL: `https://pypi.org/project/robotframework-restlibrary/`.

[CLB+17] Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 4299–4307, 2017. URL: `https://proceedings.neurips.cc/paper/2017/hash/d5e2c0adad503c91f91df240d0cd4e49-Abstract.html`.

[CP08] Gerardo Canfora and Massimiliano Di Penta. Service-oriented architectures testing: A survey. In Andrea De Lucia and Filomena Ferrucci, editors, *Software Engineering, International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, volume 5413 of *Lecture Notes in Computer Science*, pages 78–105. Springer, 2008. `doi:10.1007/978-3-540-95888-8\_4`.

[CTJ+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL: `https://arxiv.org/abs/2107.03374`, `arXiv:2107.03374`.

[CZPC22] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Resttestgen: An extensible framework for automated black-box testing of restful apis. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*, pages 504–508. IEEE, 2022. `doi:10.1109/ICSME55016.2022.00068`.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019. URL: `https://doi.org/10.18653/v1/n19-1423`, `doi:10.18653/V1/N19-1423`.

[DLMV$^+$24] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. Pentestgpt: evaluating and harnessing large language models for automated penetration testing. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, pages 847 –864, USA, 2024. USENIX Association.

[DPP$^+$24] Alix Decrop, Gilles Perrouin, Mike Papadakis, Xavier Devroey, and Pierre-Yves Schobbens. You can REST now: Automated specification inference and black-box testing of restful apis with large language models. *CoRR*, abs/2402.05102, 2024. URL: `https://doi.org/10.48550/arXiv.2402.05102`, `arXiv:2402.05102`, `doi:10.48550/ARXIV.2402.05102`.

[DVK17] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning, 2017. URL: `https://arxiv.org/abs/1702.08608`, `arXiv:1702.08608`.

[EACM22] Adeel Ehsan, Mohammed Ahmad M. E. Abuhaliqa, Cagatay Catal, and Deepti Mishra. Restful api testing methodologies: Rationale, challenges, and solution directions. *Applied Sciences*, 12(9:4369), 2022. URL: `https://www.mdpi.com/2076-3417/12/9/4369`, `doi:10.3390/app12094369`.

[FB15] Tobias Fertig and Peter Braun. Model-driven testing of restful apis. In Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015, Florence, Italy, May 18-22, 2015 - Companion Volume*, pages 1497–1502. ACM, 2015. doi:10.1145/2740908.2743045.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. Publication, University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[FM24] Ineza Felin-Michel. Web service vs. rest api: Understanding the key differences, Nov 2024. Accessed: 2025-04-01. URL: https://medium.com/towards-agi/web-service-vs-rest-api-understanding-the-key-differences-87ad5ce44693.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002. doi:10.1145/514183.514185.

[GGM+20] Luca Gazzola, Maayan Goldstein, Leonardo Mariani, Itai Segall, and Luca Ussi. Automatic ex-vivo regression testing of microservices. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, AST '20, page 11–20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387903.3389309.

[GGM+23] Luca Gazzola, Maayan Goldstein, Leonardo Mariani, Marco Mobilio, Itai Segall, Alessandro Tundo, and Luca Ussi. Exvivomicrotest: Exvivo testing of microservices. *J. Softw. Evol. Process.*, 35(4), 2023. URL: https://doi.org/10.1002/smr.2452, doi:10.1002/SMR.2452.

[GOP+23] Olga Golovneva, Sean O'Brien, Ramakanth Pasunuru, Tianlu Wang, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. Pathfinder: Guided search over multi-step reasoning paths. *CoRR*, abs/2312.05180, 2023. URL: https://doi.org/10.48550/arXiv.2312.05180, arXiv:2312.05180, doi:10.48550/ARXIV.2312.05180.

[GZA23] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. Testing restful apis: A survey. *ACM Trans. Softw. Eng. Methodol.*, 33(1), November 2023. doi:10.1145/3617175.

[HBM+22] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent

Sifre. Training compute-optimal large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, pages 30016–30030, Red Hook, NY, USA, 2022. Curran Associates Inc.

[HC23] Andreas Happe and Jürgen Cito. Getting pwn'd by AI: penetration testing with large language models. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 2082–2086. ACM, 2023. `doi:10.1145/3611643.3613083`.

[HDD22] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving semantics-aware fuzzers from web api schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 345–346, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3510454.3528637`.

[HR18] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 328–339. Association for Computational Linguistics, 2018. URL: `https://aclanthology.org/P18-1031/`, `doi:10.18653/V1/P18-1031`.

[HR24] Iryna Hartsock and Ghulam Rasool. Vision-language models for medical report generation and visual question answering: a review. *Frontiers in Artificial Intelligence*, 7, November 2024. URL: `http://dx.doi.org/10.3389/frai.2024.1430984`, `doi:10.3389/frai.2024.1430984`.

[HSAB22] API Security Protect Your APIs from Attacks Home School Academy, Mark O'Neill and Data Breaches. Api security - protect your apis from attacks and data breaches gartner 15 jul 2021, Feb 2022. Accessed: 2025-04-02. URL: `https://www.coursehero.com/file/130422560/API-Security-Protect-Your-APIs-from-Attacks-and-Data-Breaches-Gartner-15-jul-2021pdf/`.

[Hul23] Dave Hulbert. Using tree-of-thought prompting to boost chatgpt's reasoning, May 2023. URL: `https://doi.org/10.5281/zenodo.10323452`, `doi:10.5281/ZENODO.10323452`.

[Ibm24] Ibm. What is a rest api?, Dec 2024. Accessed: 2025-04-01. URL: `https://www.ibm.com/think/topics/rest-apis`.

[ISCK24] Isamu Isozaki, Manil Shrestha, Rick Console, and Edward Kim. Towards automated penetration testing: Introducing LLM benchmark, analysis, and improvements. *CoRR*, abs/2410.17141, 2024. URL: `https://doi.org/10.48550/arXiv.2410.17141`, `arXiv:2410.17141`, `doi:10.48550/ARXIV.2410.17141`.

[JLF$^+$23] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Comput. Surv.*, 55(12):248:1–248:38, 2023. `doi:10.1145/3571730`.

[JLM$^+$24] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. When fuzzing meets llms: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 492–496, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3663529.3663784`.

[JSM$^+$23] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. *CoRR*, abs/2310.06825, 2023. URL: `https://doi.org/10.48550/arXiv.2310.06825`, `arXiv:2310.06825`, `doi:10.48550/ARXIV.2310.06825`.

[KAT$^+$25] Komal Kumar, Tajamul Ashraf, Omkar Thawakar, Rao Muhammad Anwer, Hisham Cholakkal, Mubarak Shah, Ming-Hsuan Yang, Phillip H. S. Torr, Salman H. Khan, and Fahad Shahbaz Khan. LLM post-training: A deep dive into reasoning large language models. *CoRR*, abs/2502.21321, 2025. URL: `https://doi.org/10.48550/arXiv.2502.21321`, `arXiv:2502.21321`, `doi:10.48550/ARXIV.2502.21321`.

[KCS20] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. Quickrest: Property-based test generation of openapi-described restful apis. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 131–141. IEEE, 2020. `doi:10.1109/ICST46399.2020.00023`.

[KCS$^+$23] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. Enhancing rest api testing with nlp techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1232–1243, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3597926.3598131`.

[KGR+22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, pages 22199–22213, Red Hook, NY, USA, 2022. Curran Associates Inc.

[KMH+20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020. URL: `https://arxiv.org/abs/2001.08361`, `arXiv:2001.08361`.

[KSO23] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. Adaptive REST API testing with reinforcement learning. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 446–458. IEEE, 2023. `doi:10.1109/ASE56229.2023.00218`.

[KSS+24] Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro 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, NIER@ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 37–41. ACM, 2024. `doi:10.1145/3639476.3639769`.

[KXSO22] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated test generation for rest apis: no time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 289–301, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3533767.3534401`.

[Li23] Yinheng Li. A practical survey on zero-shot prompt design for in-context learning. In Ruslan Mitkov and Galia Angelova, editors, *Proceedings of the 14th International Conference on Recent Advances in Natural Language Processing, RANLP 2023, Varna, Bulgaria, 4-6 September 2023*, pages 641–647. INCOMA Ltd., Shoumen, Bulgaria, 2023. URL: `https://aclanthology.org/2023.ranlp-1.69`.

[Lim24] Magicmind Technologies Limited. Api testing: Challenges, solutions, and best practices explained, Aug 2024. Accessed: 2025-04-01. URL: `https://medium.com/@magicminds/api-testing-challenges-solutions-and-best-practices-explained-b197d0b68e97`.

[LLZ18] Jing Liu, Zhen-Tian Liu, and Yu-Qiang Zhao. CPN model based standard feature verification method for REST service architecture. In Honghao Gao, Xinheng Wang, Yuyu Yin, and Muddesar Iqbal, editors, *Collaborative Computing: Networking, Applications and Worksharing - 14th EAI International Conference, CollaborateCom 2018, Shanghai, China, December 1-3, 2018, Proceedings*, volume 268 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 688–707. Springer, 2018. doi:10.1007/978-3-030-12981-1_48.

[Lon23] Jieyi Long. Large language model guided tree-of-thought. *CoRR*, abs/2305.08291, 2023. URL: https://doi.org/10.48550/arXiv.2305.08291, arXiv:2305.08291, doi:10.48550/ARXIV.2305.08291.

[LZ25] Hao Lin and Yongjun Zhang. The risks of using large language models for text annotation in social science research, 2025. URL: https://arxiv.org/abs/2503.22040, arXiv:2503.22040.

[MASR21] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. Black-box and white-box test case generation for restful apis: Enemies or allies? In Zhi Jin, Xuandong Li, Jianwen Xiang, Leonardo Mariani, Ting Liu, Xiao Yu, and Nahgmeh Ivaki, editors, *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, pages 231–241. IEEE, 2021. doi:10.1109/ISSRE52982.2021.00034.

[MLH+22] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 11048–11064. Association for Computational Linguistics, 2022. URL: https://doi.org/10.18653/v1/2022.emnlp-main.759, doi:10.18653/V1/2022.EMNLP-MAIN.759.

[MLP+23] Ian R. McKenzie, Alexander Lyzhov, Michael Pieler, Alicia Parrish, Aaron Mueller, Ameya Prabhu, Euan McLean, Aaron Kirtland, Alexis Ross, Alisa Liu, Andrew Gritsevskiy, Daniel Wurgaft, Derik Kauffman, Gabriel Recchia, Jiacheng Liu, Joe Cavanagh, Max Weiss, Sicong Huang, The Floating Droid, Tom Tseng, Tomasz Korbak, Xudong Shen, Yuhui Zhang, Zhengping Zhou, Najoung Kim, Samuel R. Bowman, and Ethan Perez. Inverse scaling: When bigger isn't better. *Trans. Mach. Learn. Res.*, 2023, 2023. URL: https://openreview.net/forum?id=DwgRm72GQF.

[MLSRC22] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Online testing of restful apis: promises and challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on* *the Foundations of Software Engineering*, ESEC/FSE 2022, page 408–420, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3540250.3549144.

[MSR21] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In Cristian Cadar and Xiangyu Zhang, editors, *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 682–685. ACM, 2021. doi:10.1145/3460319.3469082.

[Mye04] Glenford J. Myers. *The art of software testing (2. ed.)*, pages 120–130. Wiley, 2004. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471469122.html.

[Ole25] Michał Oleszak. Reinforcement learning from human feedback (rlhf) for llms, Mar 2025. Accessed: 2025-04-01. URL: https://neptune.ai/blog/reinforcement-learning-from-human-feedback-for-llms.

[Ope23] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. URL: https://doi.org/10.48550/arXiv.2303.08774, arXiv:2303.08774, doi:10.48550/ARXIV.2303.08774.

[Ope24a] OpenAI. Gpt-4o mini: Advancing cost-efficient reasoning, 2024. Accessed: 2025-04-01. URL: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence//.

[Ope24b] OpenAI. Gpt-4o: Openai's latest model, 2024. Accessed: 2025-04-01. URL: https://www.techtarget.com/whatis/feature/GPT-4o-explained-Everything-you-need-to-know.

[Ope24c] OpenAI. Openai o1 and o1-mini: Optimized ai models for specific tasks, 2024. Accessed: 2025-04-01. URL: https://openai.com/research/introducing-openai-o1-preview/.

[PKT15] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE Computer Society, 2015. doi:10.1109/ICST.2015.7102604.

[QLY+24] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

OpenReview.net, 2024. URL: `https://openreview.net/forum?id=dHng2O0Jjr`.

[R24] Kamesh R. Think beyond size: Adaptive prompting for more effective reasoning, 2024. URL: `https://arxiv.org/abs/2410.08130`, `arXiv:2410.08130`.

[RBC+21] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew J. Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *CoRR*, abs/2112.11446, 2021. URL: `https://arxiv.org/abs/2112.11446`, `arXiv:2112.11446`.

[req23] Reqres api, 2023. Accessed: 2025-04-01. URL: `https://reqres.in/api/users/1`.

[RFBL99] Jeffrey Mogul Henrik Frystyk Larry Masinter Paul Leach Roy Fielding, Jim Gettys and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. (1999)., 1999. Accessed: 2025-04-01. URL: `https://www.rfc-editor.org/rfc/rfc2616?data1=dwnsb4B&data2=abmurltv2b`.

[RGG+23] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. URL: `https://doi.org/10.48550/arXiv.2308.12950`, `arXiv:2308.12950`, `doi:10.48550/ARXIV.2308.12950`.

[RR07] Leonard Richardson and Sam Ruby. *RESTful web services - web services for the real world*, pages 215–258. O'Reilly, 2007. URL: `http://www.oreilly.com/catalog/9780596529260/index.html`.

[RSR+20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL: `https://jmlr.org/papers/v21/20-074.html`.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019. Accessed: 2025-04-01. URL: `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`.

[SFA+22] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilic, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, Jonathan Tow, Alexander M. Rush, Stella Biderman, Albert Webson, Pawan Sasanka Ammanamanchi, Thomas Wang, Benoît Sagot, Niklas Muennighoff, Albert Villanova del Moral, Olatunji Ruwase, Rachel Bawden, Stas Bekman, Angelina McMillan-Major, Iz Beltagy, Huu Nguyen, Lucile Saulnier, Samson Tan, Pedro Ortiz Suarez, Victor Sanh, Hugo Laurençon, Yacine Jernite, Julien Launay, Margaret Mitchell, Colin Raffel, Aaron Gokaslan, Adi Simhi, Aitor Soroa, Alham Fikri Aji, Amit Alfassy, Anna Rogers, Ariel Kreisberg Nitzav, Canwen Xu, Chenghao Mou, Chris Emezue, Christopher Klamm, Colin Leong, Daniel van Strien, David Ifeoluwa Adelani, and et al. BLOOM: A 176b-parameter open-access multilingual language model. *CoRR*, abs/2211.05100, 2022. URL: `https://doi.org/10.48550/arXiv.2211.05100`, `arXiv:2211.05100`, `doi:10.48550/ARXIV.2211.05100`.

[SGM19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 3645–3650. Association for Computational Linguistics, 2019. URL: `https://doi.org/10.18653/v1/p19-1355`, `doi:10.18653/V1/P19-1355`.

[SIB+24] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson C. Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Miserlis Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompting techniques. *CoRR*, abs/2406.06608, 2024. URL: `https://doi.org/10.48550/arXiv.2406.06608`, `arXiv:2406.06608`, `doi:10.48550/ARXIV.2406.06608`.

[SIG+24] Chandan Singh, Jeevana Priya Inala, Michel Galley, Rich Caruana, and Jianfeng Gao. Rethinking interpretability in the era of large language models. *CoRR*, abs/2402.01761, 2024.

[Suf24] Fahim K. Sufi. Generative pre-trained transformer (GPT) in research: A systematic review on data augmentation. *Inf.*, 15(2):99, 2024. URL: `https://doi.org/10.3390/info15020099`, `doi:10.3390/INFO15020099`.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL: `http://arxiv.org/abs/1707.06347`, `arXiv:1707.06347`.

[SWR+22] Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal V. Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Févry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. Multitask prompted training enables zero-shot task generalization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: `https://openreview.net/forum?id=9Vrb9D0WI4`.

[SXZ+23] Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. Restgpt: Connecting large language models with real-world applications via restful apis. *CoRR*, abs/2306.06624, 2023. URL: `https://doi.org/10.48550/arXiv.2306.06624`, `arXiv:2306.06624`, `doi:10.48550/ARXIV.2306.06624`.

[Tea] Swagger Team. Open api specification. Accessed: 2025-04-01. URL: `https://swagger.io/specification/`.

[Tec24] Devstringx Technologies. 10 top challenges in rest api automation testing — devstringx, Jul 2024. Accessed: 2025-04-01. URL: `https://devstr` ingx-technologies.medium.com/10-top-challenges-in-rest-api-automation-testing-devstringx-a2cf57386cb2.

[TLI+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023. URL: https://doi.org/10.48550/arXiv.2302.13971, arXiv:2302.13971, doi:10.48550/ARXIV.2302.13971.

[TMS+23] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023. URL: https://doi.org/10.48550/arXiv.2307.09288, arXiv:2307.09288, doi:10.48550/ARXIV.2307.09288.

[TTH21] Chung-Hsuan Tsai, Shi-Chun Tsai, and Shih-Kun Huang. REST API fuzzing by coverage level guided blackbox testing. In *21st IEEE International Conference on Software Quality, Reliability and Security, QRS 2021, Hainan, China, December 6-10, 2021*, pages 291–300. IEEE, 2021. doi:10.1109/QRS54544.2021.00040.

[VIS24] Bernard Loki "AI VISIONARY". Openai o1 vs chat gpt 4o: Exploring key features and differences., 2024. Accessed: 2025-04-01. URL: https://medium.com/@bernardloki/openai-o1-vs-chat-gpt-4o-exploring-key-features-and-differences-c411671b239f#:~:text=Extended%20Thinking%20Time:%20The%20model,across%20various%20academic%20disciplines%2C%20including:.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL: `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`.

[WBZ$^+$22] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: `https://openreview.net/forum?id=gEZrGCozdqR`.

[WMR$^+$21] Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, Zac Kenton, Sasha Brown, Will Hawkins, Tom Stepleton, Courtney Biles, Abeba Birhane, Julia Haas, Laura Rimell, Lisa Anne Hendricks, William Isaac, Sean Legassick, Geoffrey Irving, and Iason Gabriel. Ethical and social risks of harm from language models. *CoRR*, abs/2112.04359, 2021. URL: `https://arxiv.org/abs/2112.04359`, `arXiv:2112.04359`.

[WWS$^+$22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, pages 24824–24837, 2022. URL: `http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html`.

[WZS$^+$23] Xinyi Wang, Wanrong Zhu, Michael Saxon, Mark Steyvers, and William Yang Wang. Large language models are latent variable models: Explaining and finding good demonstrations for in-context learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, pages 15614–15638, 2023. URL: `http://papers.nips.cc/paper_files/paper/2023/hash/3255a7554605a88800f4e120b3a929e1-Abstract-Conference.html`.

[WZX$^+$23] Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 2550–2575. Association for Computational Linguistics, 2023. URL: `https://doi.org/10.18653/v1/2023.findings-emnlp.167`, `doi:10.18653/V1/2023.FINDINGS-EMNLP.167`.

[YYZ+23] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, pages 11809–11822, 2023. URL: `http://papers.nips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html`.

[ZLM+23] Honghua Zhang, Liunian Harold Li, Tao Meng, Kai-Wei Chang, and Guy Van den Broeck. On the paradox of learning to reason from data. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, pages 3365–3373. ijcai.org, 2023. URL: `https://doi.org/10.24963/ijcai.2023/375`, `doi:10.24963/IJCAI.2023/375`.

[ZRG+22] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022. URL: `https://arxiv.org/abs/2205.01068`, `arXiv:2205.01068`.

[ZYYY23] Yifan Zhang, Jingqin Yang, Yang Yuan, and Andrew Chi-Chih Yao. Cumulative reasoning with large language models. *CoRR*, abs/2308.04371, 2023. URL: `https://doi.org/10.48550/arXiv.2308.04371`, `arXiv:2308.04371`, `doi:10.48550/ARXIV.2308.04371`.