An Intelligent Agent for Automated Test Generation from OpenAPI Specifications

Ricardo F. Vilela, Stevão Alves de Andrade, Eduardo B. F. Santos, Williamson Silva, Pedro H. D. Valle

School of Technology, University of Campinas (FT/Unicamp), Limeira, SP, Brazil

Center of Excellence in Social Technologies (NEES/UFAL), Maceió, AL, Brazil

Federal University of Cariri (UFCA), Juazeiro do Norte, CE, Brazil

University of São Paulo (IME-USP), São Paulo, SP, Brazil

`rfvilela@unicamp.br`

`{stevao.andrade, eduardo.farias}@nees.ufal.br`

`williamson.silva@ufca.edu.br, pedrohenriquevalle@usp.br`

Abstract. *Context: The widespread adoption of RESTful APIs demands effective contract validation, especially in critical domains. Motivation: Tools such as Postman and Newman automate test execution, but test creation remains manual, error-prone, and difficult to maintain. Objective: This work proposes an intelligent agent based on LLMs to interpret OpenAPI documents and automatically generate test collections. Method: The solution fragments OpenAPI documents by endpoint, generates specialized prompts, and uses an LLM via OpenRouter to create automated tests, validated with Newman on a public (anonymized) API. Results: Execution produced 34 requests, of which 12 were successful, 14 had critical failures (malformed URIs), and 8 contained structural or semantic errors. The results demonstrate feasibility and reduced manual effort, while indicating the need for further validation and refinement for CI/CD use.*

1. Introduction

Adopting RESTful APIs has become one of the cornerstones in the development of modern systems, ranging from simple web applications to complex integrations in corporate platforms and public services. This resource-oriented approach, built on HTTP operations, has become a widely accepted standard due to its flexibility, scalability, and compatibility with different technologies and programming languages [Golmohammadi et al. 2023].

With the advancement of digital transformation and the increasing interconnectivity among systems, APIs have evolved from mere technical add-ons to playing a strategic role in application architecture. They act as intermediaries for communication between different components and organizations, being fundamental, for example, in the interoperability of educational systems, the integration of healthcare services, or the control of financial transactions in government environments [Dullabh et al. 2019].

However, as the complexity of these integrations grows, so does the demand for effective mechanisms to validate the contracts established between clients and servers. Even a slight deviation in the structure or semantics of a request can compromise the application's functionality, the integrity of sensitive data, and the end-user experience [Ehsan et al. 2022].

In this context, API testing becomes an essential practice to ensure system robustness, reliability, and predictability. It is crucial to validate not only the common request flows but also the behavior in response to invalid inputs, authentication errors, access limits, and unexpected scenarios. This is especially relevant in critical domains such as education, healthcare, or public finance, where failures can have significant social impacts [Martin-Lopez et al. 2022].

Tools such as Postman[1] and Newman[2] are widely used in the testing ecosystem, enabling the execution and automation of request collections. However, creating and maintaining these collections is still essentially a manual process and is prone to errors. Frequent changes in API specifications demand constant rewriting of tests, which increases operational costs and may compromise standardization and coverage.

In this scenario, intelligent agents have the potential to support repetitive and technically complex activities within the software lifecycle. Unlike conventional automated scripts or bots, intelligent agents are autonomous systems capable of interpreting complex inputs—such as OpenAPI[3] documents—making decisions based on contextual knowledge, and acting in the digital environment with a certain degree of autonomy [Erlenhov et al. 2020]. Although the use of Large Language Models (LLMs) has advanced in tasks such as code generation and review assistance, their application to the intelligent generation of RESTful API tests is still at an early stage [Nooyens et al. 2025, Kim et al. 2024].

This paper presents the development of an intelligent agent specifically designed to automate the entire technical process of generating functional tests for RESTful APIs from OpenAPI documents. The proposed solution integrates AI techniques with a web-based interface, implementing a well-defined processing pipeline—from specification validation, endpoint fragmentation, and prompt generation to the assembly of Postman-compatible test collections—ensuring a reproducible and technically sound workflow.

The goal is to investigate how an AI-powered interpretative agent can reduce the manual effort required for test creation while promoting reproducibility, standardization, and faster adaptation to API changes. To evaluate the approach, the agent was applied to the public RESTful API of the Pé-de-Meia Program (Brazilian Ministry of Education – MEC), representative of a production environment and used nationwide for managing student benefit eligibility, enabling an in-depth observation of its operation and limitations.

The main contributions of this work include:

- Development of an intelligent agent with a web interface to automate the generation of functional tests from OpenAPI documents through a complete, structured

[1]https://www.postman.com/
[2]https://www.npmjs.com/package/newman
[3]https://www.openapis.org/

pipeline;
- Introduction of a route-fragmentation strategy combined with specialized prompts to overcome LLM context limitations;
- Practical evaluation of the solution on a real-world API, demonstrating its applicability for supporting software quality assurance.

The remainder of this paper is organized as follows: Section 2 presents the theoretical and technological foundations of the approach; Section 3 provides a detailed description of the architecture and implementation of the agent; Section 4 presents the tests performed and the results obtained; Section 5 discusses the challenges encountered and potential improvements; and Section 6 presents conclusions and directions for future work.

2. Background

The development of intelligent agents is a classical area within Artificial Intelligence (AI). According to Russell and Norvig [Russell and Norvig 2009], an intelligent agent is any entity capable of perceiving its environment through sensors and acting upon it via actuators, to maximize some performance measure. These agents differ from purely reactive systems due to their ability to reason, plan, and adapt to changing contexts.

Wooldridge [Balaji and Srinivasan 2010] complements this definition by emphasizing that intelligent agents operate autonomously, controlling their state and actions without direct human intervention. They are typically characterized by proactive behavior (taking the initiative), reactive behavior (responding to the environment), and social behavior (interacting with other agents or humans).

In software engineering, agents have been applied to tasks such as code generation, requirements analysis, and automated testing [Santhanam et al. 2022]. Software testing is critical for ensuring system reliability, particularly in high-stakes domains [Ammann and Offutt 2016], and literature highlights the importance of combining formal techniques with empirical practices for adequate software validation [Graham et al. 2008].

Test automation for RESTful APIs has gained significant attention due to their widespread adoption in the software industry. According to Fielding [Fielding 2000], RESTful APIs follow architectural principles that promote scalability, simplicity, and adherence to web standards such as the HTTP protocol, URIs, and standardized methods [Fielding and Reschke 2014]. API contracts are commonly described using the OpenAPI specification [Miller et al. 2021], enabling the formal documentation of endpoints, parameters, and expected responses.

Despite advances in specification, test creation remains largely manual. Studies such as those by Arcuri [Arcuri 2019, Arcuri et al. 2024], Kim et al. [Kim et al. 2022], and Golmohammadi et al. [Golmohammadi et al. 2023] point to the need for tools that automate test generation from these contracts. Strategies range from random testing [Duran and Ntafos 1984], to search-based techniques [McMinn 2004, Ferreira Vilela et al. 2023], and intelligent fuzzing [Arcuri et al. 2025].

In this context, Large Language Models (LLMs) introduce new possibilities. Trained on vast corpora of textual data, LLMs can perform tasks such as interpretation, synthesis, and content generation. Recent studies, such as Pearce et al.

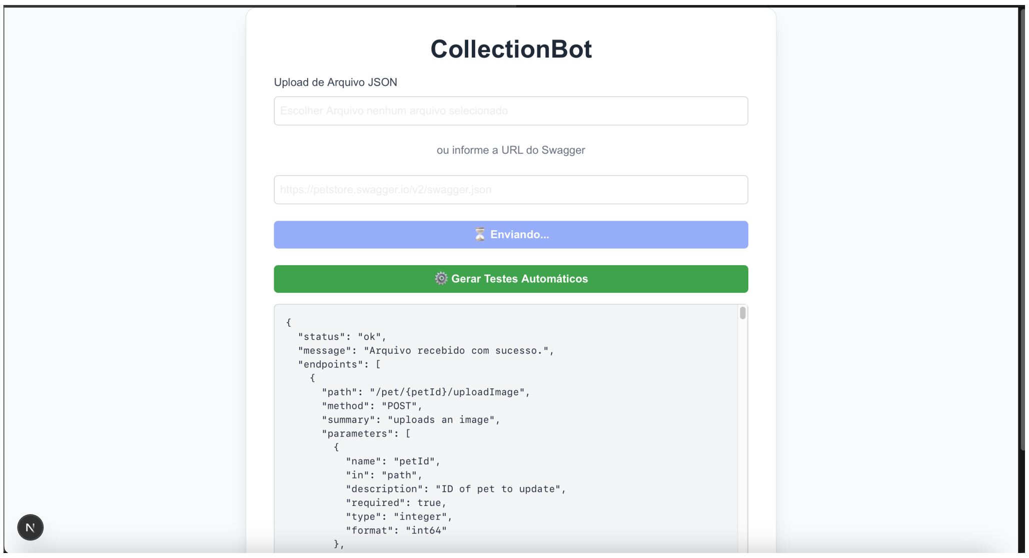
[Pearce et al. 2025], highlight the use of LLMs in development environments through tools like GitHub Copilot. While risks related to insecure code generation remain, LLMs have acted as co-agents, enhancing automation throughout the development and testing lifecycle.

This work builds upon this background by employing an intelligent agent powered by LLMs, capable of interpreting OpenAPI documents and generating functional test collections. By automating the transformation of API contracts into executable scripts with realistic validations, the agent contributes to improving software quality and reducing the manual effort required in the testing phase [Martin-Lopez et al. 2019, Marculescu et al. 2022].

3. Solution Description

The developed solution comprises an intelligent agent accessible through an intuitive web interface. Its architecture follows a modular design, organized into two main layers:

Frontend, built with Next.js, is responsible for user interaction, file uploads, and displaying the generated results. Backend, implemented in Node.js, is responsible for processing the API specification, communicating with the Large Language Model (LLM), and assembling the resulting test collection. The workflow begins when the user provides an OpenAPI specification. This can be done either by uploading a JSON file or by supplying a URL that points to a Swagger[4] document. Figure 1 illustrates this initial interaction step.


CollectionBot
Upload de Arquivo JSON
Escolher Arquivo nenhum arquivo selecionado
ou informe a URL do Swagger
https://petstore.swagger.io/v2/swagger.json
⌛ Enviando...
⚙️ Gerar Testes Automáticos
N

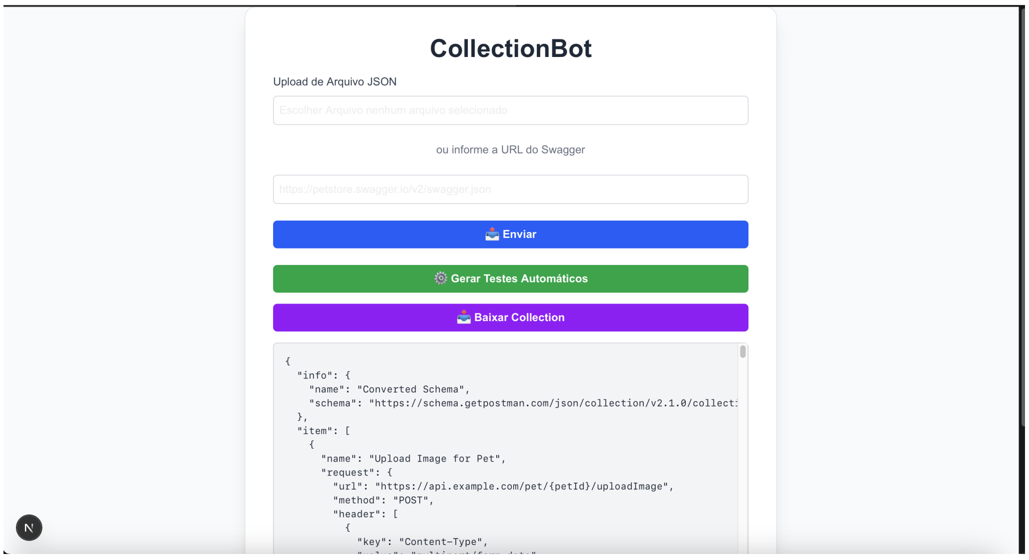
```
{
  "status": "ok",
  "message": "Arquivo recebido com sucesso.",
  "endpoints": [
    {
      "path": "/pet/{petId}/uploadImage",
      "method": "POST",
      "summary": "uploads an image",
      "parameters": [
        {
          "name": "petId",
          "in": "path",
          "description": "ID of pet to update",
          "required": true,
          "type": "integer",
          "format": "int64"
        },
```

Figura 1. Upload interface for an OpenAPI specification and automatic test generation.

Once received, the specification undergoes structural validation to ensure it conforms to the OpenAPI schema. It is then fragmented by endpoint, allowing each API route to be processed individually. For each fragment, the system constructs a specialized prompt that combines:

[4]Swagger is a framework for describing and visualizing RESTful APIs. It includes the OpenAPI specification and a set of tools for API design, documentation, and testing.

The technical description of the route (method, URL, parameters, request body, and expected responses); Explicit instructions for generating a Postman-compatible test case, including assertions for HTTP status codes, response payload structure, and negative test cases. These prompts are sent to an LLM through the OpenRouter API[5], which acts as an intermediary between the backend and different available language models. The LLM's response contains the JSON structure for an individual test item in the Postman Collection format. The backend progressively builds the collection, aggregating all generated test cases. The generated tests include realistic values for query parameters, request bodies, and HTTP headers, ensuring that the test scenarios closely resemble real-world usage. Once complete, the collection is made available for download, as shown in Figure 2.


CollectionBot
Upload de Arquivo JSON
Escolher Arquivo nenhum arquivo selecionado
ou informe a URL do Swagger
https://petstore.swagger.io/v2/swagger.json
Enviar
Gerar Testes Automáticos
Baixar Collection

```
{
  "info": {
    "name": "Converted Schema",
    "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collecti
  },
  "item": [
    {
      "name": "Upload Image for Pet",
      "request": {
        "url": "https://api.example.com/pet/{petId}/uploadImage",
        "method": "POST",
        "header": [
          {
            "key": "Content-Type",
```

Figura 2. Interface after generating the collection, with download option available.

The generated collection is fully compatible with the Newman[6] command-line tool, enabling execution either locally or within Continuous Integration (CI) pipelines. During execution, results are automatically evaluated according to predefined criteria. If the collection fails to meet minimum quality thresholds—such as having at least two validation assertions per endpoint—the system triggers an iterative refinement loop, adjusting prompts or optimizing inputs until the criteria are satisfied.

The complete system workflow, from specification ingestion to test generation and refinement, is summarized in Figure 3.

4. Practical Evaluation and Results

To assess the applicability of the proposed solution, we conducted a proof of concept using a publicly available RESTful API, extensively documented in OpenAPI and composed of dozens of endpoints. We conducted a proof of concept using the public RESTful API of the "Pé-de-Meia" Program, provided by the Brazilian Ministry of Education

[5]OpenRouter is a gateway that allows access to multiple LLM providers using a unified API.
[6]Newman is a command-line tool for executing Postman collections, often used in automation pipelines.

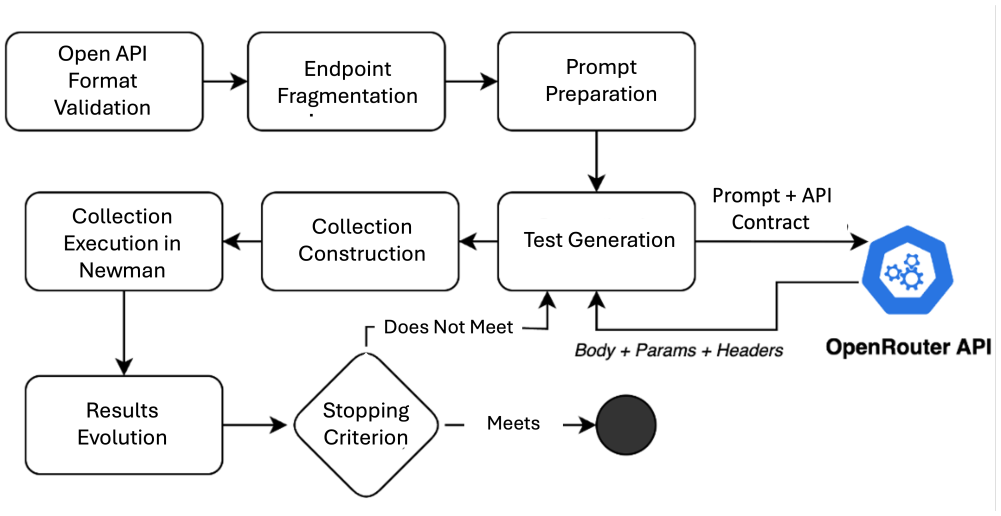


Figura 3. Workflow of the intelligent agent for automated test generation from OpenAPI specifications.

(MEC) and available at `https://api-cmde.gestaopresente.mec.gov.br/v1/documentation`. This API is documented in OpenAPI format and is composed of dozens of endpoints covering functionalities such as student registration, benefit management, and reporting.

Test Collection Generation

We submitted the OpenAPI specification to the intelligent agent, which processed each route individually. As a result, the agent automatically generated 34 requests, covering `GET`, `POST`, and `PATCH` operations. No `DELETE` operations were generated, as such endpoints were not present in the target API due to business-specific constraints.

The generated collection followed the Postman Collection v2.1 format and included:

- Authentication headers such as `x-api-key`, predefined by the user;
- Simulated input data (JSON payloads, path parameters, and query strings);
- Automated verification scripts, such as `pm.response.to.have.status(200)`.

Execution and Evaluation with Newman

To evaluate the executability of the generated tests, we used Newman in a controlled environment. The collection was executed with the following command, employing an environment variable to simulate the API's actual domain:

```
newman run collection_<timestamp>.postman_collection.json
--env-var "url=https://api-cmde.gestaopresente.mec.gov.br/v1/"
```

Listing 1. Newman command for executing the generated collection

During execution, Newman processed all 34 collection items. Table 1 summarizes the execution results:

The most recurrent errors observed during execution were:

Tabela 1. Summary of test collection execution

Category	Count
Successful requests (status 200 or expected 400)	12
Critical execution failures (invalid URI)	14
Semantic or structural errors	8

1. **Missing base domain:** the `raw` field was omitted in several requests, producing malformed URIs such as `http:///v1/entities`.
2. **Persistence of placeholder domains:** some requests retained the domain `https://api.example.com`, ignoring the provided environment variable.
3. **Unresolved placeholders:** parameters such as `id` (e.g., `12345`), `cpf` (e.g., `12345678900`), and `inep` (e.g., `987654321`) were not replaced, leading to `404 Not Found` or `400 Bad Request`.
4. **Missing request body in `POST` or `PATCH` methods:** resulting in `422 Unprocessable Entity` responses.

These results show that, although automated generation succeeded for part of the test cases, the model still has limitations when adapting generated artifacts for execution in a real environment. URL assembly errors, missing mandatory data, and unresolved placeholders indicate the need for post-processing and validation mechanisms.

On the other hand, the correct generation of complete request structures—including headers, payloads, and verification scripts—demonstrates the agent's ability to interpret and reproduce expected patterns from the OpenAPI specification. Table 2 summarizes the main execution failures and their probable causes.

Tabela 2. Main failures during test execution

Failure Type	Probable Cause
Invalid URI	Missing `raw` field or malformed domain
Unexpected 404/400	Unreplaced placeholders or missing parameters
Error 422	Missing or incorrect request body
Placeholder domain	Prompt ignored or misinterpreted by the model

These findings highlight the need for complementary strategies such as prompt enrichment, automated syntactic validation, and iterative refinement based on test execution feedback. Such mechanisms could significantly improve the reliability of the generated collections in production-like environments.

5. Discussion

The conducted evaluation shows that applying Large Language Models (LLMs) to generate functional tests from OpenAPI specifications automatically is both feasible and promising. The proposed approach demonstrated the ability to transform formal endpoint descriptions into executable artifacts compatible with widely adopted professional tools such as Postman and Newman.

One of the main benefits observed was the significant reduction in manual effort required to build test collections. Automating the generation of authentication headers, input examples, and validation scripts contributed not only to process efficiency but also to request standardization—an essential factor in projects involving multiple developers or distributed teams.

However, the results also revealed relevant limitations. The most notable was the unpredictability of LLM behavior when processing complex or poorly specified inputs. Even with explicit instructions, the model frequently failed to replace placeholder domains, fill mandatory parameters, or assemble complete URLs—issues that rendered a considerable portion of the generated tests non-executable.

Another critical point relates to the high sensitivity of results to prompt engineering. Slight variations in instruction wording produced significant discrepancies in the generated artifacts' structure, indicating a lack of deterministic consistency. This undermines process reliability in environments where repeatability and strict version control are required.

From an architectural perspective, the adopted endpoint fragmentation strategy proved effective in addressing LLM context size limitations. However, it introduced a new challenge: consolidating the generated fragments into a cohesive final collection, which may require post-processing to ensure structural and semantic completeness.

Execution through Newman played a central role in the evaluation, enabling the identification of both syntactic errors (e.g., malformed URIs) and semantic issues (e.g., unexpected responses). These findings reinforce the need for integrating automated verification and enrichment mechanisms after generation, especially in Continuous Integration (CI) contexts.

In Summary, while the operational results are not yet fully satisfactory, the proposed approach demonstrates strong potential as a complementary tool in the quality assurance process. By automating repetitive and error-prone tasks, the intelligent agent can increase the productivity of both development and QA teams. Its modular architecture also supports incremental evolution, allowing for fine-tuning of prompts, model replacement, and the incorporation of verification heuristics.

This work positions itself as an initial contribution to the use of LLMs for contract-driven automation, paving the way for more robust, hybrid solutions integrated into the modern DevOps and software engineering ecosystem.

6. Conclusion and Future Work

This work demonstrated the feasibility of employing AI-based Large Language Models (LLMs) to automate the generation of tests for RESTful APIs from OpenAPI documents. The proposed solution, implemented as an intelligent agent with a web interface, was able to produce enriched Postman test collections, reducing manual effort, promoting standardization, and enabling integration with automated execution tools such as Newman.

The practical proof of concept showed that, despite challenges related to model variability, computational cost, and the need for prompt refinement, the approach is functional and applicable to real-world contexts. The iterative cycle guided by stopping criteria proved effective in improving the quality of the generated tests.

For future work, we plan to extend the solution to support additional test formats, such as JMeter and k6, and to integrate it into CI/CD platforms. We also intend to explore alternative LLMs and hybrid modeling strategies, as well as implement mechanisms for automatic prompt adjustment based on execution feedback.

Referências

Ammann, P. and Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press, 2 edition.

Arcuri, A. (2019). Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):1–37.

Arcuri, A., Poth, A., and Rrjolli, O. (2025). Introducing black-box fuzz testing for rest apis in industry: Challenges and solutions. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

Arcuri, A., Zhang, M., and Galeotti, J. (2024). Advanced white-box heuristics for search-based fuzzing of rest apis. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–36.

Balaji, P. G. and Srinivasan, D. (2010). *An Introduction to Multi-Agent Systems*, pages 1–27. Springer Berlin Heidelberg, Berlin, Heidelberg.

Dullabh, P., Hovey, L., Heaney-Huls, K., Rajendran, N., Wright, A., and Sittig, D. F. (2019). Application programming interfaces (apis) in health care: findings from a current-state assessment. In *Context Sensitive Health Informatics: Sustainability in Dynamic Ecosystems*, pages 201–206. IOS Press.

Duran, J. W. and Ntafos, S. C. (1984). An evaluation of random testing. *IEEE transactions on Software Engineering*, (4):438–444.

Ehsan, A., Abuhaliqa, M. A. M. E., Catal, C., and Mishra, D. (2022). Restful api testing methodologies: Rationale, challenges, and solution directions. *Applied Sciences*, 12(9).

Erlenhov, L., Neto, F. G. d. O., and Leitner, P. (2020). An empirical study of bots in software development: characteristics and challenges from a practitioner's perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 445–455, New York, NY, USA. Association for Computing Machinery.

Ferreira Vilela, R., Choma Neto, J., Santiago Costa Pinto, V. H., Lopes de Souza, P. S., and do Rocio Senger de Souza, S. (2023). Bio-inspired optimization to support the test data generation of concurrent software. *Concurrency and Computation: Practice and Experience*, 35(2):e7489.

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

Fielding, R. T. and Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231.

Golmohammadi, A., Zhang, M., and Arcuri, A. (2023). Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–41.

Graham, D., Veenendaal, E. v., Evans, I., and Black, R. (2008). *Foundations of software testing: ISTQB certification*. Intl Thomson Business Pr.

Kim, M., Stennett, T., Shah, D., Sinha, S., and Orso, A. (2024). 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*, ICSE-NIER'24, page 37–41, New York, NY, USA. Association for Computing Machinery.

Kim, M., Xin, Q., Sinha, S., and Orso, A. (2022). 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*, pages 289–301.

Marculescu, B., Zhang, M., and Arcuri, A. (2022). On the faults found in rest apis by automated test generation. *ACM Trans. Softw. Eng. Methodol.*, 31(3).

Martin-Lopez, A., Segura, S., and Ruiz-Cortés, A. (2019). Test coverage criteria for restful web apis. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 15–21.

Martin-Lopez, A., Segura, S., and Ruiz-Cortés, A. (2022). 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. Association for Computing Machinery.

McMinn, P. (2004). Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156.

Miller, D., Whitlocak, J., Gartiner, M., Ralphson, M., Ratovsky, R., and Sarid, U. (2021). OpenAPI specification v3.1.0.

Nooyens, R., Bardakci, T., Beyazit, M., and Demeyer, S. (2025). Test amplification for rest apis via single and multi-agent llm systems.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2025). Asleep at the keyboard? assessing the security of github copilot's code contributions. *Commun. ACM*, 68(2):96–105.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition.

Santhanam, S., Hecking, T., Schreiber, A., and Wagner, S. (2022). Bots in software engineering: a systematic mapping study. *PeerJ Computer Science*, 8:e866.