

**EFFECTIVE AUTOMATION OF BLACK-BOX TESTING FOR REST APIS
WITH MACHINE LEARNING AND LANGUAGE MODELS**

A Dissertation
Presented to
The Academic Faculty

By

Myeongsoo Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

May 2025

© Myeongsoo Kim 2025

**EFFECTIVE AUTOMATION OF BLACK-BOX TESTING FOR REST APIS
WITH MACHINE LEARNING AND LANGUAGE MODELS**

Thesis committee:

Dr. Alessandro Orso
Computer Science
Georgia Institute of Technology

Dr. Spencer Rugaber
Computer Science
Georgia Institute of Technology

Dr. Manish Motwani
Electrical Engineering and Computer Sci-
ence
Oregon State University

Dr. Saurabh Sinha
AI for Code
IBM Research

Dr. Qirun Zhang
Computer Science
Georgia Institute of Technology

Date approved: April 23, 2025

ACKNOWLEDGMENTS

I would like to extend my sincerest gratitude to everyone who has contributed to the completion of this thesis proposal. This research would not have been possible without the guidance, support, and encouragement of many individuals.

First and foremost, I am deeply thankful to my thesis advisor, Dr. Alessandro Orso, for his invaluable guidance, expertise, and patience throughout this journey. His insightful feedback and advice have been essential in shaping this work. During the first two years of my PhD, while working on different projects, I struggled with motivation and engagement in my research. However, Dr. Orso continued to provide me with various research directions, notably introducing me to the field of REST API testing, which reignited my interest. Through his mentorship, I regained my passion for research and chose to continue on this path. Dr. Orso's unwavering support, from our regular meetings to his critiques of my work, has been instrumental in my growth as a researcher.

I am also grateful to my thesis committee members—Dr. Manish Motwani, Dr. Spencer Rugaber, Dr. Saurabh Sinha, and Dr. Qirun Zhang—for their time, assistance, and thoughtful contributions to this work.

- Dr. Manish Motwani, during his post-doctoral period in our lab, which coincided with Dr. Orso's busy time as interim dean of our college, served as my mentor and friend.
- Dr. Spencer Rugaber provided the most detailed feedback from my qualifier exam to the end of my PhD journey. His contributions had one of the biggest impacts on the quality of my thesis.
- Dr. Saurabh Sinha has been an inspiring collaborator since my first-year internship. Our five-year collaboration, spanning from that initial internship to the completion of my PhD, has significantly shaped my research skills and perspective. His mentorship

and expertise have been invaluable to my growth as a researcher.

- Dr. Qirun Zhang taught me how to analyze existing research and build upon it, especially during my first year, with great patience. Through two classes and two papers with him, he instilled in me valuable skills in reading and critically analyzing research papers, which have been fundamental to my development as a researcher.

I have truly learned a lot from all my committee members, and I feel fortunate to have had their guidance.

I am deeply grateful to my advisors during my undergraduate and exchange student periods. Dr. Eun-Jin Im at Kookmin University introduced me to research and gave me my first lab opportunity. At UC Irvine, Dr. Ian Harris and Dr. André van der Hoek sparked my interest in software engineering through the Undergraduate Research Opportunities Program (UROP). Their early mentorship set me on this path, for which I am truly thankful.

I would like to express my heartfelt appreciation to my collaborators and lab mates for their unwavering support, valuable discussions, and thoughtful feedback throughout my doctoral journey. Our interactions, both formal and informal, have enriched my research experience and contributed significantly to this work. I extend special thanks to my friends, particularly Jinho Hah, Seongyoung Kang, Seulki Kim, Seung Hwan Kim, Seunghyun Kim, Dohyun Ku, Hyun Jun Lee, and Mingu Kwak for their warm encouragement and support. Also, I am grateful to all my family members, including Hyewon Kim, Jinhee Kim, and Mihee Kim.

A special thanks to my wife, Hyunseo Kim, for her constant love, support, and patience. Her belief in me and her understanding have been a tremendous source of strength and motivation, without which this work would not have been possible. Finally, I extend my gratitude to all those whose names may not appear here but who have played a part in this endeavor.

TABLE OF CONTENTS

Acknowledgments	iii
List of Tables	xii
List of Figures	xiv
Summary	xvi
Chapter 1: Introduction	1
1.1 Problem Statement	2
1.2 A Framework for REST API Testing Enhancement	3
1.2.1 Empirical Study of Current Testing Approaches	4
1.2.2 Specification Enhancement for REST API Testing	4
1.2.3 Advanced Testing Methodologies with Machine Learning and Language Models	5
1.3 Research Framework and Contributions	5
1.3.1 RQ1: Natural Language Processing for API Documentation	6
1.3.2 RQ2: Large Language Models for Test Input Generation	7
1.3.3 RQ3: Reinforcement Learning for Adaptive Testing	8
1.3.4 RQ4: Specialized Language Models for Efficient Testing	9
1.3.5 RQ5: Multi-Agent Approaches for Testing Coordination	10

1.3.6	Answering the Central Research Question	11
1.4	Organization of the Dissertation	12
Chapter 2:	Background on REST APIs	14
2.1	Background	14
2.1.1	REST APIs	14
2.1.2	OpenAPI Specifications	15
2.1.3	Large Language Models	16
2.1.4	LLM Prompting Techniques	18
2.1.5	Reinforcement Learning and Q-Learning	18
2.1.6	Multi-Agent Reinforcement Learning	19
2.2	Related Work	20
2.2.1	REST API Testing Techniques	20
2.2.2	Enhancing OpenAPI specifications and test generation via NLP techniques	21
2.2.3	Requirements based test case generation	21
2.2.4	Reinforcement learning based test case generation	22
2.2.5	LLM-based REST API Testing and Analysis	23
Chapter 3:	State of the Art	24
3.1	Introduction	24
3.2	Testing Tools Used In The Study	27
3.2.1	Tools Selection	27
3.2.2	Collected Testing Tools	28

3.3	Empirical Study	31
3.3.1	Web Services Benchmark	33
3.3.2	Experiment Setup	34
3.3.3	RQ1: Coverage Achieved	35
3.3.4	RQ2: Error Responses Triggered	38
3.3.5	RQ3: Implications of the Results	40
3.3.6	Threats to Validity	48
3.4	Conclusion	50
Chapter 4: Enhancing REST API Testing with NLP techniques		51
4.1	Introduction	51
4.2	Preliminary Study	55
4.3	Approach	56
4.3.1	Approach Overview	56
4.3.2	Terminology	58
4.3.3	NLP-based Rule Extraction	59
4.3.4	Rule Validation	66
4.4	Evaluation	70
4.4.1	Research Questions	70
4.4.2	Experiment Setup	72
4.4.3	Experiment Results	74
4.4.4	Threats to Validity	79
4.5	Conclusion	80

Chapter 5: Leveraging Large Language Models to Improve REST API Testing	82
5.1 Introduction	82
5.2 Motivating Example	83
5.3 Approach	85
5.3.1 Overview	85
5.3.2 Rule Generator	86
5.3.3 Specification Enhancement	88
5.4 Preliminary Results	89
5.4.1 Evaluation Methodology	89
5.4.2 Results and Discussion	90
5.4.3 Threats to Validity	90
5.5 Conclusion	91
Chapter 6: Adaptive REST API Testing with Reinforcement Learning	93
6.1 Introduction	93
6.2 Motivating Example	97
6.3 Approach	99
6.3.1 Q-Learning Table Initialization	101
6.3.2 Q-Learning-based Prioritization	103
6.3.3 Q-Learning-based API Testing	107
6.4 Evaluation	108
6.4.1 Experiment Setup	109
6.4.2 RQ1: Effectiveness	112

6.4.3	RQ2: Efficiency	113
6.4.4	RQ3: Fault-Detection Capability	115
6.4.5	RQ4: Ablation Study	116
6.4.6	Threats to Validity	117
6.5	Conclusion	118
Chapter 7: Effective REST API Testing with Small Language Models		119
7.1	Introduction	119
7.2	Approach	122
7.2.1	Base Model Selection	122
7.2.2	Database Construction	123
7.2.3	Fine-Tuning and Quantization	125
7.2.4	REST API Testing	128
7.3	Evaluation	130
7.3.1	Experiment Setup	131
7.3.2	RQ1: Comparison with RESTGPT	132
7.3.3	RQ2: Impact of Fine-Tuning and Quantization on REST API Testing	136
7.3.4	RQ3: Comparison with State-of-the-art REST API Testing Tools . .	140
7.3.5	RQ4: Fault-Detection Capability	141
7.3.6	RQ5: Ablation Study	143
7.3.7	Threats to Validity	144
7.4	Conclusion	145

Chapter 8: A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs	147
8.1 Introduction	147
8.2 Motivating Example	150
8.3 Approach	151
8.3.1 Overview	151
8.3.2 Q-Learning and Agent Policy	153
8.3.3 Semantic Property Dependency Graph	156
8.3.4 REST Agents	158
8.3.5 Request Generator	162
8.4 Evaluation	163
8.4.1 Experiment Setup	163
8.4.2 RQ1: Effectiveness	165
8.4.3 RQ2: Fault Detection Capability	167
8.4.4 RQ3: Ablation Study	169
8.4.5 Threats to Validity	171
8.5 Conclusion	171
Chapter 9: Conclusion And Future Work	173
9.1 Summary of Contributions	173
9.1.1 Empirical Understanding of API Testing Limitations	173
9.1.2 Specification Enhancement Approaches	174
9.1.3 Advanced Testing Methodologies	174
9.2 Addressing the Research Questions	175

9.2.1	Main Research Question	175
9.2.2	Specific Research Questions	176
9.3	Impact and Benefits	179
9.4	Reflections on Tools and Methodology	179
9.5	Merit of Research	180
9.6	Limitations and Challenges	181
9.7	Directions for Future Research	182
9.7.1	Enhancing REST API Testing Techniques	182
9.7.2	Advanced Code Generation from API Specifications	183
9.7.3	Cross-Pollination with Unit Testing	184
9.7.4	Broader Applications	184
	References	186

LIST OF TABLES

3.1	Overview of REST API testing techniques and tools.	26
3.2	RESTful web services used in the empirical study.	33
3.3	Average line, branch, and method coverage achieved and 500 errors found in one hour (E: unique error, UFP: unique failure point, ULFP: unique library failure point).	35
4.1	Types of rules identified from natural-language descriptions in API specifications in the preliminary study.	55
4.2	Effectiveness of NLP-based Rule Extraction and Rule Validation.	71
4.3	Improvement in performance of the testing tools considered when fed with enhanced specifications, compared to their baseline performance with original specifications.	77
5.1	Effectiveness of NLP2REST and RestGPT.	85
5.2	Accuracy of ARTE and RestGPT.	85
6.1	Comparison of operations covered and valid and failure-inducing requests generated (2xx and 500 status codes) by the tools.	111
6.2	Total faults detected by the tools over 10 runs.	114
6.3	Results of the Ablation Study.	116
7.1	REST services used in the evaluation.	131

7.2	Comparison of semantically valid parameter value generation by LlamaREST-EX, Llama3-8B, and RESTGPT ('FT' indicates fine-tuning and 'Q' indicates quantization).	133
7.3	Comparison of inter-parameter dependency rule generation performance among LlamaREST-IPD, RESTGPT, and Llama3-8B ('FT' indicates fine-tuning, 'Q' indicates quantization, 'O' indicates correct rule generation, and '-' indicates failure to generate the rule).	133
7.4	Average number of operations processed in 10 runs for closed-source services using LlamaRestTest configured with Llama3-8B (Base), fine-tuning (FT), and quantization (Q).	136
7.5	Average number of operations processed for closed-source services by different testing tools (LlamaRestTest configured with 8-bit quantization.)	137
7.6	Internal server errors detected over 10 runs.	142
7.7	Ablation study on the impact of key components of LlamaRestTest on coverage achieved.	143
8.1	REST services used in the evaluation.	164
8.2	Number of operations exercised.	166
8.3	Service failures triggered (500 response codes).	167
8.4	Code coverage achieved by different tool variants.	169

LIST OF FIGURES

1.1	Progression of the research approaches developed in this dissertation. The research begins with identifying limitations through an empirical study, then develops methods that build upon each other. The approaches evolve from NLP-based extraction (NLP2REST) to large language model integration (RestGPT), then to reinforcement learning (ARAT-RL), specialized language models (LlamaRestTest), and finally multi-agent systems (AutoRestTest).	3
2.1	An example OpenAPI specification.	15
3.1	Code coverage achieved and number of unique 500 errors, unique failure points, and unique library failure points detected over all services by the ten tools in 10 minutes, 1 hour, and 24 hours (1: EvoMasterWB, 2: RESTler, 3: RestTestGen, 4: RESTest, 5: bBOXRT , 6: Schemathesis, 7: Tcases, 8: Dredd, 9: EvoMasterBB, 10: APIFuzzer).	35
3.2	A method used for parsing the file suffix in the SCS web service.	41
3.3	Parsed dependency graphs of <code>preferredVariants</code> parameter description (top) and <code>language</code> parameter description (bottom) from Languagetool's OpenAPI specification.	43
3.4	Sample code from the SCS web service.	45
3.5	A method of Features-Service used to find configuration names associated with a product.	46
4.1	Sample OpenAPI specification fragments to illustrate rule extraction from human-readable descriptions.	52
4.2	Overview of NLP2REST approach.	56
4.3	An example of a constituency parse tree.	61

4.4	Comparison of branch, line, and method coverage results for original (Org) and enhanced (Enh) specifications.	77
5.1	A part of FDIC Bank Data’s OpenAPI specification.	83
5.2	Approach Overview.	84
6.1	A Part of Features Service’s OpenAPI Specification.	97
6.2	Approach Overview	99
6.3	Branch, line, and method coverage achieved by the tools on the benchmark services.	108
7.1	Overview of the approach.	122
7.2	Code coverage achieved on 10 runs for open-source services using LlamaRestTest configured with the base Llama3-8B model, the fine-tuned model (FT), and the quantized models (Q).	136
7.3	Code coverage achieved on 10 runs for open-source services with MoRest, RESTler, ARAT-RL, EvoMaster, and LlamaRestTest (fine-tuned with 8-bit quantization).	139
8.1	A part of Market API’s OpenAPI Specification.	151
8.2	Overview of AutoRestTest approach.	152
8.3	Illustration of SPDG construction and refinement.	154
8.4	Comparison of code coverage metrics across tools and services: line, branch, and method coverage.	165

SUMMARY

This dissertation improves the field of automated black-box REST API testing through novel methodologies that address limitations in current approaches. As REST APIs have become the backbone of modern microservices architectures and web applications, effective testing has grown increasingly vital. Existing tools demonstrate significant shortcomings in leveraging API specifications, identifying dependencies between operations and parameters, and generating effective test inputs. This research presents a series of innovative solutions that progressively enhance testing effectiveness, coverage, and fault detection capabilities in complex REST API ecosystems.

The research begins with a comprehensive analysis of ten state-of-the-art REST API testing tools across 20 RESTful web services. This analysis revealed critical inefficiencies in three main areas: finding API operation dependencies, identifying inter-parameter dependencies, and generating effective test input values. These findings underscore the need for more sophisticated approaches to improve overall testing effectiveness.

Building on these insights, the first major contribution is a novel methodology leveraging Natural Language Processing (NLP) techniques to extract valuable semantic information from human-readable parts of OpenAPI specifications. This approach bridges the gap between human-readable documentation and machine-readable specifications, enabling more effective automated testing. Using a custom Word2Vec model with constituency parse tree analysis, the methodology transforms natural language into structured testing constraints. This enhances test generation by identifying implicit parameter constraints, revealing inter-parameter dependencies, and enabling realistic test inputs. A prototype implementation demonstrates the methodology's effectiveness across diverse API specifications.

The second major contribution advances API testing through the application of Large Language Models (LLMs) with few-shot learning capabilities. This innovation extracts im-

PLICIT operational constraints, complex parameter relationships, and semantic dependencies hidden in traditional approaches. By leveraging contextual understanding in LLMs, the methodology creates comprehensive testing artifacts that capture nuanced API behaviors without extensive training data. A proof-of-concept implementation demonstrates how this approach effectively augments OpenAPI Specifications with enriched knowledge, addressing limitations in earlier methods while generating semantically appropriate test inputs.

The third major contribution shifts the focus to adaptive testing strategies through an innovative methodology based on reinforcement learning. This approach employs Q-learning algorithms to dynamically prioritize API operations and parameters based on execution feedback, addressing the challenge of redundant test inputs common in existing tools. By integrating feedback from API requests and responses, the methodology enables efficient exploration of API behavior even with incomplete specifications. A prototype implementation demonstrates how this reinforcement learning approach significantly improves test coverage and fault detection through feedback-driven adaptive testing strategies.

The fourth major contribution advances API testing through a methodology that focuses on fine-tuned and quantized small-language models. This approach uniquely incorporates dynamic natural language from server responses to refine test inputs and rules during execution. By balancing model size and accuracy, the methodology enables efficient deployment in resource-constrained environments while maintaining high performance. The results demonstrate how specialized language models enhance API testing capabilities while utilizing natural language elements in server responses to guide the testing process.

The final contribution presents an advanced methodology integrating Multi-Agent Reinforcement Learning (MARL) with a Semantic Property Dependency Graph (SPDG) and Large Language Models. This approach addresses the challenge of vast API search spaces through collaborative specialized agents for dependencies, operations, parameters, and values. By optimizing these testing steps simultaneously, the methodology achieves higher efficiency in test generation. The implementation demonstrates how SPDG reduces search

space complexity while temporal Q-learning enables dynamic prioritization, representing an advancement in creating efficient testing strategies for complex APIs.

In conclusion, this dissertation presents a series of innovative methodologies that significantly advance automated black-box REST API testing. Each contribution builds upon previous ones, addressing critical limitations in existing approaches and expanding testing capabilities. By leveraging cutting-edge technologies in language models and machine learning, this research establishes a foundation for more efficient and effective testing practices in increasingly complex REST API ecosystems.

CHAPTER 1

INTRODUCTION

In the realm of digital interactions and services, REST (Representational State Transfer) APIs play a critical role, enabling seamless communication and data exchanges across diverse software platforms [1, 2]. REST, a paradigm pioneered by Roy Fielding, underscores principles such as scalability, flexibility, and statelessness, thereby facilitating the integration of various services without requiring deep mutual knowledge between client and server beyond what is defined in the API itself [3]. This architectural style is widely adopted in modern software engineering, exemplified by a vast array of services documented on platforms like APIs Guru [4] and Rapid API [5]. These platforms not only showcase the prevalence of REST APIs but also highlight the necessity of clear, structured documentation provided by specifications such as OpenAPI [6], formerly known as Swagger [7]. These specifications, by outlining available operations, data formats, and response types, provide a clear, structured framework that not only aids developers in understanding and accessing APIs but also enhances the compatibility with software tools designed for improving API reliability and performance. The integration of RESTful web services with OpenAPI specifications has significantly facilitated the development of numerous black-box testing tools, designed to automate the generation of test cases [8, 9, 10, 11, 12, 13, 14, 15, 16, 17].

However, while these tools have evolved to automate and streamline the testing process, they are not without limitations. Automated REST API testing tools often struggle with achieving extensive code coverage and effectively exploring the REST APIs which include API dependencies, inter-parameter dependencies, and parameter constraints. The tools typically employ a variety of testing strategies including Breadth-First Search (BFS), Depth-First Search (DFS), random walks [18], combinatorial testing, model-based search, and evolutionary algorithms. Yet, these approaches frequently fall short in effectively ex-

ploring the vast search space of API behaviors, especially in scenarios with incomplete or vague parameter and response schemas [19, 20]. The limitations in current testing methodologies underscore a crucial gap, revealing the necessity for more advanced testing mechanisms that can adapt to the complexities of modern RESTful APIs.

1.1 Problem Statement

Despite advances in automated REST API testing, existing black-box techniques face fundamental challenges that limit their effectiveness in real-world scenarios. Current approaches rely on static information from API specifications, combined with search-based algorithms that lack the adaptability to navigate complex API behaviors effectively. This dissertation addresses three critical limitations in the state of the art:

First, conventional REST API testing tools struggle to generate contextually appropriate test inputs, particularly for domain-specific parameters that require semantic understanding. Without this capability, tests frequently produce invalid requests that fail to explore meaningful API behaviors or trigger relevant faults.

Second, existing approaches have limited ability to identify and leverage dependencies between API operations and parameters. These dependencies—often implicit or documented only in natural language descriptions—create complex preconditions for successful API interactions that current tools cannot systematically discover or utilize.

Third, the search space for meaningful API test sequences grows exponentially with API size, making exhaustive exploration infeasible. Current search strategies (random, evolutionary, or combinatorial) lack the intelligence to prioritize promising paths effectively, resulting in inefficient testing that misses critical API behaviors.

This dissertation hypothesizes that modern machine learning techniques—particularly natural language processing, large language models, and reinforcement learning—can address these limitations by:

1. Extracting implicit knowledge from natural language API documentation

2. Generating semantically valid test inputs tailored to specific REST API contexts
3. Learning adaptive testing strategies that effectively navigates REST API operation dependencies
4. Developing collaborative multi-agent approaches that specialize in different aspects of the testing process

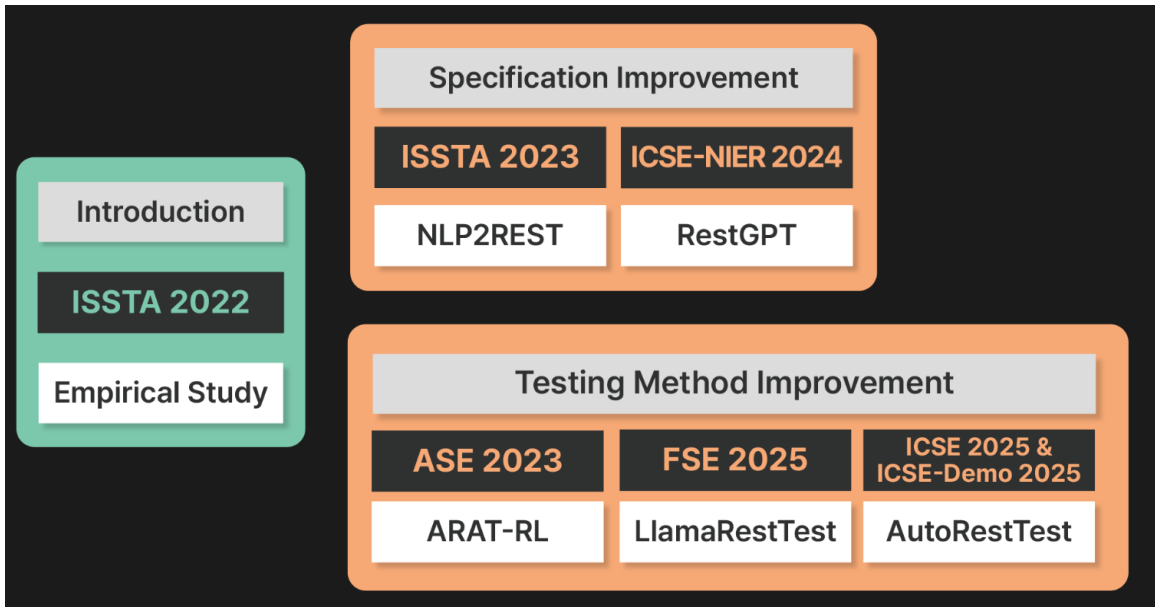


Figure 1.1: Progression of the research approaches developed in this dissertation. The research begins with identifying limitations through an empirical study, then develops methods that build upon each other. The approaches evolve from NLP-based extraction (NLP2REST) to large language model integration (RestGPT), then to reinforcement learning (ARAT-RL), specialized language models (LlamaRestTest), and finally multi-agent systems (AutoRestTest).

1.2 A Framework for REST API Testing Enhancement

This research begins with an empirical study identifying three key limitations in existing REST API testing approaches: ineffective input generation, overlooked dependencies, and inefficient search space exploration. To address these challenges, as shown in Figure 1.1, I first developed NLP2REST, which applies natural language processing to extract

valuable information from API documentation. Building on insights from this work, Rest-GPT leverages large language models to more accurately identify constraints and generate contextually appropriate test values. Moving beyond specification enhancement, ARAT-RL introduces reinforcement learning for adaptive testing strategies, which LlamaRestTest further refines by incorporating specialized, efficient language models. The research culminates in AutoRestTest, which employs a multi-agent approach that combines the strengths of all previous techniques while adding semantic dependency modeling. Each approach progressively addresses more complex aspects of the testing challenge, with later methods building upon concepts and techniques established in earlier work.

1.2.1 Empirical Study of Current Testing Approaches

First, I conducted a large-scale empirical study [19] comparing 10 state-of-the-art REST API testing tools on a benchmark of 20 real-world RESTful web services. This evaluation—based on code coverage achieved and unique failures triggered—revealed critical limitations in existing approaches:

1. Difficulty in generating valid input values, especially for domain-specific parameters
2. Limitations in detecting dependencies between API parameters
3. Inefficiencies in exploring the vast search space of API dependencies

These findings provided the foundation for subsequent contributions by identifying specific areas where improvement was most needed.

1.2.2 Specification Enhancement for REST API Testing

Second, I developed techniques to enhance API specifications by extracting information from natural language documentation that traditional tools overlook. This contribution focuses on bridging the gap between human-readable documentation and machine-readable specifications through:

1. Natural language processing techniques to extract parameter constraints, dependencies, and examples (NLP2REST, Chapter 4)
2. Large language models to more accurately identify rules and generate contextually appropriate parameter values (RestGPT, Chapter 5)

These approaches extract and formalize the human-readable information in API specifications, enabling existing testing tools to generate more effective test cases without modifying their core algorithms.

1.2.3 Advanced Testing Methodologies with Machine Learning and Language Models

Third, I explored how various machine learning techniques and language models can improve the testing methodology itself, moving beyond specification enhancement to create more intelligent and adaptive testing approaches:

1. Reinforcement learning for adaptive test case prioritization and execution (ARAT-RL, Chapter 6)
2. Fine-tuned small language models for resource-efficient testing (LlamaRestTest, Chapter 7)
3. Multi-agent reinforcement learning for collaborative API exploration (AutoRestTest, Chapter 8)

This progression demonstrates how different machine learning and language models can address increasingly complex challenges in REST API testing, with each approach building on insights gained from previous ones.

1.3 Research Framework and Contributions

This dissertation is guided by one central research question.

Question: Can machine learning techniques and language models enhance the effectiveness of black-box REST API testing by increasing code and API coverage, improving fault detection, and addressing limitations of traditional approaches?

To systematically address this complex question, I decomposed it into five specific research questions, each exploring a different aspect of how machine learning can improve REST API testing:

1. Can natural language processing extract useful information from OpenAPI Specification to improve REST API testing? (RQ1)
2. Can large language models improve test input generation and parameter constraint identification in REST API testing? (RQ2)
3. Can reinforcement learning be applied to create adaptive testing strategies that efficiently explore the REST APIs? (RQ3)
4. What benefits can specialized, fine-tuned language models bring to REST API testing in terms of both effectiveness and computational efficiency? (RQ4)
5. Can a multi-agent approach coordinate specialized testing components to improve the effectiveness of REST API testing? (RQ5)

Each of these questions is addressed in a dedicated chapter with specific approaches and findings as outlined below.

1.3.1 RQ1: Natural Language Processing for API Documentation

Approach: This work introduces Natural Language Processing (NLP) techniques to extract valuable information from human-readable parts of OpenAPI specifications for REST API testing. Prior to this work, automated testing tools largely ignored the rich information contained in natural language descriptions within API documentation - content that humans can easily interpret but machines typically cannot. This approach focuses on identifying

parameter type/format, inter-parameter dependencies, parameter constraints, and examples from these textual descriptions.

The approach employs a custom Word2Vec model trained on a corpus of OpenAPI specifications and related documentation, enabling it to capture domain-specific semantic relationships. The use of constituency parse trees allows for a deeper understanding of sentence structure, facilitating more accurate extraction of complex constraints and dependencies. A two-phase process of rule extraction and validation ensures the accuracy of the extracted information. The rule extraction phase identifies potential rules, while the validation phase uses both static analysis and dynamic checking against the actual API implementation to refine and verify these rules.

Answer: Yes, as demonstrated in Chapter 4, NLP techniques can successfully extract valuable testing information from API documentation. Experimental evaluations with NLP2REST demonstrated substantial improvements in test generation, including increases in successful API requests, code coverage, and unique server error responses detected. By bridging the gap between human-readable documentation and machine-readable specifications, this approach enhanced the effectiveness of existing testing tools by enriching OpenAPI specifications with previously untapped information.

1.3.2 RQ2: Large Language Models for Test Input Generation

Approach: This work uses the power of Large Language Models (LLMs) to extract rules and generate contextually relevant parameter values from API specifications. This approach addresses limitations of previous techniques by utilizing advanced natural language understanding and generation capabilities to process both machine-readable and human-readable components of OpenAPI specifications.

The methodology employs carefully crafted prompts, including guidelines, specific cases, grammar highlights, and output configurations, to instruct LLMs on how to interpret API documentation and generate appropriate rules and values. The approach focuses

on extracting four types of rules: operational constraints, parameter constraints, parameter type and format, and parameter examples. Unlike NLP-based approaches that require validation phases to improve precision, this LLM-based methodology achieves high accuracy without expensive validation processes due to the superior ability to understand context and nuance in natural language descriptions.

Answer: Yes, as detailed in Chapter 5, Large Language Models can significantly improve constraint identification and test input generation. Preliminary evaluations showed that RestGPT outperformed previous approaches such as NLP2REST in precision and ARTE in generating valid inputs. By leveraging LLMs' advanced capabilities, RestGPT effectively enriches API specifications with valuable information that can be used by existing testing tools to generate more effective test cases, improving code coverage and fault detection accuracy.

1.3.3 RQ3: Reinforcement Learning for Adaptive Testing

Approach: This work introduces a novel approach that leverages reinforcement learning to enhance REST API testing. The methodology addresses the limitations of previous techniques by employing an adaptive testing strategy that intelligently prioritizes operations and parameters during the API exploration process.

The core of this approach is a Q-learning based algorithm that dynamically adjusts the importance of API operations and parameters based on feedback received from the API. This allows for focused testing on areas of the API that are more likely to increase code coverage, efficiently navigating the vast search space of possible API interactions. Unlike previous approaches that treat all operations and parameters equally, this reinforcement learning methodology optimizes the testing process through experience-driven exploration.

Another key feature of this methodology is its dynamic construction of key-value pairs from both request and response data, allowing for the identification of hidden dependencies and resources that may not be evident from the API specification alone. Additionally, the

approach employs a sampling-based strategy for processing API feedback, enabling more efficient exploration of the API's behavior.

Answer: Yes, as demonstrated in Chapter 6, reinforcement learning can create effective adaptive testing strategies. Experimental evaluations showed ARAT-RL consistently demonstrating superior performance when compared to state-of-the-art tools like RESTler, EvoMaster, and Morest across various metrics, including branch, line, and method coverage, as well as the number of faults detected. This research demonstrates the potential of applying machine learning techniques to the challenging domain of API testing, paving the way for more effective adaptive testing strategies in the future.

1.3.4 RQ4: Specialized Language Models for Efficient Testing

Approach: This work introduces an approach that leverages small language models to enhance REST API testing. The methodology addresses the limitations of previous techniques by employing fine-tuned and quantized language models to dynamically generate test inputs and identify inter-parameter dependencies during the testing process.

The core of this approach consists of two key components: a language model for identifying inter-parameter dependencies (LlamaREST-IPD) and another for generating input values for operation parameters (LlamaREST-EX). These models are created by fine-tuning a base language model (Llama3-8b) for their specific tasks. The dependency identification model is trained using a carefully constructed IPD database containing 551 parameters with documented inter-parameter dependencies, while the value generation model is trained on an extensive EX database of over 1.8 million parameters mined from APIs-guru. To improve efficiency, both models underwent quantization, reducing their size and computational requirements while maintaining high performance.

A notable feature of this methodology is its ability to incorporate server feedback into the testing process. Building upon the reinforcement learning foundation established in ARAT-RL, LlamaRestTest integrates these specialized language models to dynamically

adapt its testing strategy. When testing encounters difficulties processing an API operation, the language models utilize both parameter descriptions and server response messages to provide dependency rules and example values.

Answer: Yes, as shown in Chapter 7, specialized language models can improve testing effectiveness while maintaining computational efficiency. In comparative evaluations, LlamaRestTest demonstrated substantial improvements over existing state-of-the-art tools in terms of code coverage, operation coverage, and fault detection capability. This research demonstrates the potential of using smaller, more efficient language models to achieve state-of-the-art performance in API testing, striking an optimal balance between effectiveness and computational efficiency while building upon the reinforcement learning based framework established in previous work.

1.3.5 RQ5: Multi-Agent Approaches for Testing Coordination

Approach: This work introduces a framework that employs a multi-agent approach to enhance REST API testing. The methodology addresses limitations of existing techniques by integrating Multi-Agent Reinforcement Learning (MARL) with a Semantic Property Dependency Graph (SPDG) and Large Language Models (LLMs).

The core innovation of this approach lies in treating REST API testing as a separable problem where four specialized agents—API, dependency, parameter, and value—collaborate to optimize API exploration. This multi-agent structure allows for a more nuanced and effective testing strategy, as each agent can focus on optimizing its specific aspect of the testing process.

A key component of this methodology is the SPDG, which simplifies the search space for dependencies using a similarity score between API operations. This graph structure enables more effective identification and utilization of relationships between different API operations, guiding the selection of dependent operations for API requests. Additionally, the approach incorporates LLMs to handle domain-specific value restrictions, generating

more realistic and contextually appropriate parameter values that improve the overall quality of test cases.

The MARL framework employs value decomposition techniques that enable these agents to efficiently coordinate their actions while learning from shared experiences. Through value decomposition, the system appropriately distributes rewards to each agent based on their contribution to successful API interactions, allowing them to make independent decisions while maintaining a cohesive overall testing strategy.

Answer: Yes, as demonstrated in Chapter 8, a multi-agent approach can effectively coordinate specialized testing components. Experimental evaluations demonstrated AutoRestTest’s effectiveness in combining the strengths of multi-agent reinforcement learning, semantic modeling of API dependencies, and the capabilities of large language models. The approach significantly outperformed state-of-the-art tools in terms of code coverage, operation coverage, and fault detection, including being the only tool able to trigger certain internal server errors in production systems. This research represents a significant advancement in REST API testing by demonstrating how specialized agents can collaborate to tackle the complex challenge of API testing.

1.3.6 Answering the Central Research Question

After systematically investigating each research question, we can now answer our central question: Can machine learning techniques and language models enhance black-box REST API testing effectiveness?

Answer: Yes, machine learning techniques and language models significantly enhance REST API testing effectiveness. Our empirical results demonstrate consistent improvements in code coverage, API operation coverage, and fault detection compared to traditional approaches. The progression from NLP-based extraction (NLP2REST) to large language models (RestGPT) to reinforcement learning (ARAT-RL) to specialized language models (LlamaRestTest) and finally to multi-agent systems (AutoRestTest) demonstrates

increasingly sophisticated ways machine learning can address the limitations of conventional black-box testing methods:

1. **Input Generation Challenge:** Language models can generate semantically valid inputs that respect domain-specific constraints, significantly reducing invalid requests.
2. **Dependency Detection Challenge:** The semantic dependency graph and language models successfully identify both explicit and implicit dependencies between operations and parameters, enabling more effective test sequence generation.
3. **Search Space Navigation Challenge:** Reinforcement learning and multi-agent approaches provide adaptive strategies that efficiently explore the vast API behavior space, focusing on promising paths.

These results confirm that machine learning offers substantial improvements to REST API testing, not merely as incremental enhancements but as fundamental shifts in how testing can be approached. The techniques developed in this dissertation collectively represent a new paradigm for API testing that leverages AI to overcome limitations that have persisted in traditional approaches for years.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

- **Chapter 2** presents background information and terminology related to REST API testing, establishing the foundational concepts necessary for understanding the research.
- **Chapter 3** provides an extensive empirical study on state-of-the-art REST API testing tools, identifying their strengths, limitations, and opportunities for improvement.

- **Chapter 4** introduces the NLP-based Rule Extraction Method (NLP2REST), detailing how natural language processing techniques can extract valuable rules from API documentation.
- **Chapter 5** presents the LLM-based Rule Extraction and Value Generation approach (RestGPT), explaining how Large Language Models can enhance specification understanding and test input generation for API testing.
- **Chapter 6** describes Adaptive REST API Testing with Reinforcement Learning (ARAT-RL), elaborating on how Q-learning can dynamically adjust testing strategies based on API feedback.
- **Chapter 7** discusses Language Model-based REST API Testing (LlamaRestTest), demonstrating the effectiveness of combining reinforcement learning with fine-tuned, resource-efficient language models.
- **Chapter 8** details the Multi-Agent Approach for REST API Testing (AutoRestTest), explaining how specialized collaborative agents can improve API coverage and fault detection.
- **Chapter 9** concludes the dissertation by summarizing the key findings, discussing limitations, and outlining directions for future research.

CHAPTER 2

BACKGROUND ON REST APIS

2.1 Background

2.1.1 REST APIs

REST APIs are web APIs that follow the RESTful architectural style [3]. Clients can communicate with web services through their REST APIs by sending HTTP requests to the services and receiving responses. Clients send requests to access and/or manipulate resources managed by the service, where a *resource* represents data that a client may want to create, delete, or access. The request is sent to an *API endpoint*, which is identified by a resource path, together with an *HTTP method* that specifies the action to be performed on the resource. The most commonly used methods are POST, GET, PUT, and DELETE, which are used to create, read, update, and delete a resource, respectively. The combination of an endpoint plus an HTTP method is called an *operation*. In addition to specifying an operation, a request can optionally also specify HTTP headers containing metadata (e.g., the data format of the resource targeted) and a body that contains the payload for the request (e.g., text in JSON format containing the input values).

After receiving and processing a request, the web service returns a response that includes, in addition to headers and possibly a body, an HTTP status code—a three-digit number that shows the outcome of the request. Status codes are organized into five suitably numbered groups. 1xx codes are used for provisional purposes, indicating the ongoing processing of a request. 2xx codes indicate successful processing of a request. 3xx codes indicate redirection and show, for instance, that the target resource has moved to a new URL (code 301). 4xx codes indicate client errors. For example, a 404 code indicates that the client has requested a resource that does not exist. Finally, 5xx codes indicate server

```

1  "/products/{productName}": {           12    }],
2  "get": {                               13    "responses": {
3    "operationId":                       14      "200": {
4      "getProductByName",                15        "description":
5    "produces":                           16          "successful operation",
6      ["application/json"],              17        "schema": {
7    "parameters": [{                     18          "$ref":
8      "name": "productName",              19          "#/definitions/Product"
9      "in": "path",                       20        },
10     "required": true,                    21     "headers": {}
11     "type": "string"                     22   }}}

```

Figure 2.1: An example OpenAPI specification.

errors in processing the request. Among this group, in particular, the 500 code indicates an Internal Server Error and typically corresponds to cases in which the service contains a bug and failed to process a request correctly. For this reason, many empirical studies of REST API testing tools report the number of 500 status codes triggered as the bugs they found (e.g., [8, 9, 10, 11, 12, 13, 14, 15, 16, 17]).

2.1.2 OpenAPI Specifications

Service interfaces are commonly documented through specifications that enumerate available operations along with their corresponding input parameters, output data types, and potential response codes. The OpenAPI Specification [6] (formerly Swagger) represents a widely adopted standard format for such documentation. Alternative specification languages include RAML [21] and API Blueprint [22], which serve similar purposes.

Figure 2.1 shows a fragment of the OpenAPI specification for Features-Service, one of the benchmarks in this study. It specifies an API endpoint `/products/{productName}` (line 1), that supports HTTP method GET (line 2). The id for this operation is `getProductByName` (lines 3–4), and the format of the returned data is JSON (lines 5–6). To exercise this endpoint, the client *must* provide a value for the required parameter `productName` in the form of a path parameter (lines 7–12) (e.g., a request `GET /products/p` where `p` is the product name). If a product with that name exists and the service can process the request correctly, the client would receive a response with status

code 200 (line 14) and the requested product data. Lines 18–19 reference the definition of the product data type, which contains information about the product, such as its id, name, features, and constraints (I omit the definition for lack of space).

Two operations have a *producer-consumer* dependency relationship when one of the operations (producer) can return data needed as input by the other operation (consumer). For example, operation `GET /products/{productName}` has producer-consumer relationship with operation `DELETE /products/{productName}/constraints/{constraintId}` (not shown in the figure). This is because a request sent to the former can lead to a response containing a constraint id needed to make a request to the latter.

2.1.3 Large Language Models

Large Language Models (LLMs), such as the Generative Pre-trained Transformer (GPT) series, are at the forefront of advancements in natural language processing (NLP) [23]. A language model, in essence, is a type of artificial intelligence that understands, interprets, and generates text in a way that is similar to how humans use language [24]. These models are trained on extensive collections of written text, allowing them to learn a wide range of linguistic patterns and styles.

The GPT series, including well-known models like GPT-3, are examples of these advanced LLMs [25]. Trained on diverse and vast datasets, they have shown remarkable skill in producing text that is strikingly human-like, making them valuable for a variety of applications. This breakthrough has broadened the horizons of NLP, leading to its application in challenging areas ranging from education to customer service [26, 27].

GPT-3 and its successors demonstrate the effectiveness of LLMs, particularly in their versatility to handle a wide array of language-related tasks [24]. These tasks can be as simple as writing coherent text or as complex as answering questions and translating languages. This versatility is a testament to their advanced capabilities, making them instrumental in pushing the boundaries of both theoretical and practical aspects of language technology [28,

29, 24].

Quantization of Large Language Models

Quantization is a technique used to reduce the computational resources required while minimizing the loss in model accuracy [30]. This process involves converting the standard floating-point representations of a model's weights and activations into lower-precision formats, such as 16-bit or 8-bit, or even lower [31, 32]. The primary advantage of quantization is the significant reduction in memory usage and computational demand, which allows these models to be deployed more efficiently [33]. The implementation of quantization can be performed in several ways, including post-training quantization and quantization-aware training. Post-training quantization involves applying quantization techniques after a model has been fully trained, which is simpler to implement and allows users to select various options depending on their resources, but it might lead to higher degradation in accuracy and efficiency. Quantization-aware training, on the other hand, simulates lower precision during the training process itself, which can help mitigate the loss in model accuracy as the model learns to adapt to the quantization effects [34].

Fine-Tuning of Large Language Models

Fine-tuning is a technique for adapting pre-trained Large Language Models (LLMs) to specific tasks, allowing them to perform better on specialized datasets by improving their prediction accuracy. A notable advancement in this area is the introduction of Lora (Low-rank Adaptation of Large Language Models), which modifies only a small set of parameters during training, specifically the attention mechanism's low-rank components. This approach minimizes the computational overhead and reduces the number of parameters updated, facilitating efficient fine-tuning of large models [35]. Building on the principles of Lora, QLora (Quantized Low-rank Adaptation) incorporates quantization into the low-rank adaptation process to further decrease the memory and computational demands. By

reducing the precision of the low-rank factors, QLoRA maintains model accuracy while enhancing computational efficiency, making it ideal for deployment in resource-constrained environments [36].

2.1.4 LLM Prompting Techniques

Prompting techniques have significantly transformed the application of LLMs in complex problem-solving and decision-making tasks [37]. Prompting essentially involves providing LLMs with contextual information or a series of instructions, guiding their responses and leveraging their inherent language understanding and generation capabilities for specific tasks [38]. A notable advancement in this domain is the Chain-of-Thought Prompting, which structures prompts to facilitate a quasi-reasoning process within LLMs, thereby considerably enhancing their problem-solving abilities [39]. Another innovative technique is Iterative Bootstrapping, which refines LLM reasoning by incorporating the model's previous responses into new prompts [40]. This iterative approach creates a feedback loop, enabling the model to build on its past outputs, leading to a more nuanced understanding and sophisticated response generation.

2.1.5 Reinforcement Learning and Q-Learning

Reinforcement learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment [41]. The agent selects actions in various situations (states), observes the consequences (rewards), and learns to choose the best actions to maximize the cumulative reward over time. The learning process in RL is trial-and-error based, meaning the agent discovers the best actions by trying out different options and refining its strategy based on the observed rewards. The agent must also decide between exploring new actions to gather more knowledge or exploiting known actions that offer the best reward based on its current understanding. The balance between exploration and exploitation is often governed by parameters, such as ϵ in the ϵ -greedy strategy [41].

Q-learning is a widely used model-free reinforcement learning algorithm that estimates the optimal action-value function, $Q(s, a)$ [42]. The Q-function represents the expected cumulative reward the agent can obtain by taking action a in state s and then following the optimal policy. Q-learning uses a table to store Q-values and updates them iteratively based on the agent’s experiences. In the learning process, the agent takes actions, receives rewards, and updates the Q-values using the Q-learning update rule, derived from the Bellman equation [41]:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (2.1)$$

where α is the learning rate, γ is the discount factor, s' is the new state after taking action a , and r is the immediate reward received. The agent updates the Q-values to converge to their optimal values, which represent the expected long-term reward of taking each action in each state.

2.1.6 Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning (MARL) extends the concepts of single-agent reinforcement learning to scenarios involving multiple interacting agents. Each agent in MARL simultaneously learns its policy while interacting not only with the environment but also with other agents [43]. This interaction can lead to complex dynamics because the agents’ actions can affect the state of the environment and the rewards of other agents. MARL can be cooperative, competitive, or mixed, depending on the nature of the interactions between the agents.

In cooperative MARL, agents work together towards a common goal, such as optimizing a shared reward function. A key challenge in this setting is credit assignment, where agents must discern their individual contributions to the collective outcome [44]. Competitive MARL, on the other hand, involves agents whose goals are typically at odds, leading them to learn strategies that outperform or block their opponents.

A significant aspect of MARL research focuses on developing mechanisms for effective communication and coordination among agents. Learning to communicate, in particular, involves developing protocols that allow agents to share relevant information to improve their decision-making processes. Furthermore, scalability remains a critical challenge due to the exponential growth in the number of interactions and the state space as the number of agents increases.

2.2 Related Work

2.2.1 REST API Testing Techniques

Automated testing for REST APIs has seen various strategies. White-box and black-box approaches, such as those in EvoMaster, leverage evolutionary algorithms to refine test cases dynamically, with a focus on internal server errors (500 status code) detection [45]. Black-box methodologies, exemplified by RESTler [9], generate stateful tests by deducing producer-consumer dependencies, aiming at detecting server failures. RestTestGen [8] and similar tools exploit data dependencies and employ oracles for validating status codes and response schemas. MoRest [17] focuses on model-based testing to simulate user interactions and test case generation, while RestCT [14] employs combinatorial testing techniques to systematically explore parameter value combinations and their effects on API behavior. ARAT-RL [15] introduces a reinforcement learning approach, dynamically refining API testing strategies by adapting to real-time feedback. Techniques like QuickREST [10], Schemathesis [13], and RESTest [11] use property-based testing and various oracles to ensure response compliance with OpenAPI or GraphQL specifications. Tools such as Dredd [46], fuzz-lightyear [47], and Tcases [16] offer diverse testing capabilities, from comparing expected responses to detecting vulnerabilities and validating Swagger-based specifications. More recently, DeepREST [48] has been proposed. DeepREST uses deep reinforcement learning to discover implicit API constraints, employing a single agent that learns operation orderings through a reward mechanism.

2.2.2 Enhancing OpenAPI specifications and test generation via NLP techniques

Several approaches have attempted to leverage natural language processing to improve REST API testing, though they differ significantly in scope and methodology from my contributions. RestCT [14] generates automated REST tests by inferring inter-parameter dependencies through pattern matching. It uses a combinatorial testing approach to handle complex APIs and employs ACTS [49], a popular covering array generation tool, to generate constrained covering arrays. However, RestCT relies on simple pattern matching rather than sophisticated NLP techniques to process API documentation.

ARTE [50] represents a more NLP-focused approach that generates test inputs for web APIs by querying the DBpedia knowledge base [51] using parameter names identified from parameter descriptions. While innovative, ARTE has significant limitations compared to our work: it focuses solely on generating example input values and does not extract or utilize constraints provided in human-readable specifications, such as default, minimum, and maximum values for parameters.

Other research has explored related but distinct applications of NLP in the API domain. Some techniques [52, 53] extract OpenAPI specifications from semi-structured web API documentation using machine learning approaches, focusing on specification generation rather than test enhancement. Broader applications of NLP for testing [54, 55, 56, 57, 58, 59, 60] have explored generating tests from program specifications, though generally not in the context of REST APIs specifically.

2.2.3 Requirements based test case generation

While this thesis focuses specifically on REST API testing, it shares conceptual foundations with broader requirements-based testing approaches that extract testable information from natural language specifications. Requirements-based testing techniques have evolved various strategies for deriving test cases from specifications. Early approaches like uc-sCNL [61] introduced controlled natural language for use case specifications, providing

structure to natural language requirements to facilitate test generation. Similarly, UML Collaboration Diagrams [62] offered a visual approach to requirements specification from which tests could be derived.

More formalized approaches emerged with Requirements by Contracts [63], which proposed a custom language for functional requirements, and SCENT-Method [64], which employed a scenario-based approach with statecharts. These techniques focused on transforming semi-structured requirements into more formal representations for testing.

Other notable contributions include SDL-based Test Generation [65], which transformed SDL requirements into Extended Finite State Machines (EFSMs), and RTCM [66], which introduced a natural language-based framework with templates, rules, and keywords to facilitate requirements-to-test transformation.

2.2.4 Reinforcement learning based test case generation

Reinforcement learning has emerged as a promising approach for automated software testing, though primarily in domains other than REST API testing. Unlike web and mobile application testing where hidden states present significant challenges, REST API testing benefits from explicit API specifications but faces different complexities in operation sequencing, parameter selection, and value generation.

Several pioneering works have applied reinforcement learning to web application testing. Zheng et al. [67] developed a curiosity-driven reinforcement learning approach that effectively explores web client behaviors without predefined test scripts. This approach shares conceptual similarities with our methodology, though applied to a different problem domain.

The majority of reinforcement learning applications in testing have focused on mobile applications, particularly Android. Koroglu et al. [68] presented QBE, a Q-learning-based exploration method specifically designed for Android apps. Similarly, Pan et al. [69] and Adamo et al. [70] developed reinforcement learning techniques for Android GUI testing,

with the former emphasizing curiosity-driven exploration. Mariani et al. [71] proposed Au-toBlackTest for black-box testing of interactive applications, while Vuong and Takada [72] and Koroğlu and Sen [73] further extended reinforcement learning approaches for Android application testing.

2.2.5 LLM-based REST API Testing and Analysis

Large Language Models (LLMs) have recently emerged as powerful tools for REST API testing and analysis, though current approaches often emphasize API understanding rather than comprehensive testing methodologies. Several notable contributions highlight both the potential and limitations of existing LLM-based approaches.

NESTFUL [74] provides an important benchmark for evaluating LLMs on nested API calls, revealing significant challenges in handling complex API interactions. This work demonstrates that while LLMs have improved API comprehension, they still struggle with intricate dependencies and multi-step operations—challenges our methodology specifically addresses through structured dependency analysis and reinforcement learning.

RESTSpecIT [75] demonstrates automated specification inference and black-box testing capabilities with minimal user input, representing a step toward more accessible API testing. However, its testing functionality is limited to basic parameter mutation without the sophisticated dependency identification and test generation strategies our approach employs.

Song et al.’s RestGPT [76] (distinct from the similarly named contribution) introduces coarse-to-fine online planning for improved task decomposition and API selection. While this work focuses primarily on API usage rather than testing, it demonstrates how LLMs can effectively reason about API operations—a capability I extend specifically for testing purposes.

CHAPTER 3

STATE OF THE ART

3.1 Introduction

The last decade has seen a tremendous growth in the availability of web APIs—APIs that provide access to a service through a web interface. This increased popularity has been driven by various industry trends, including the advent of cloud computing, the broad adoption of microservices [77], and newer value propositions enabled by the “API economy”. A majority of modern web APIs adhere to the REpresentational State Transfer (REST) architectural style [3] and are referred to as RESTful APIs, whose popularity is reflected in the availability of thousands of APIs in public directories (e.g., RapidAPI [5] and APIs guru [4]).

Given the large number of applications that rely on web APIs, it is essential to test these APIs thoroughly to ensure their quality. It is therefore not surprising that many automated techniques and tools for REST APIs have been proposed in recent years [78, 45, 9, 79, 80, 10, 81, 82, 83, 84, 85, 86, 87, 88]. These techniques take as input a description of the API, in the OpenAPI specification format [6] or API Blueprint [22], and employ various strategies to generate test cases for exercising API endpoints defined in the specification.

Although these tools have been evaluated, these evaluations have been performed (1) in different settings (in terms of API benchmarks considered, experiment setup, and metrics used), (2) using benchmarks that are in some cases limited in size or closed-source, and (3) mostly in isolation or involving only limited comparisons. It is thus difficult to understand how these tools compare to one another. Recently, and concurrently to this effort, there has been some progress in this direction, via two empirical comparisons of black-box REST API testing tools [89, 90] and an evaluation of white-box and black-box REST API test

generation [91]. These efforts are a step in the right direction but are still limited in scope. Specifically, and to the best of my knowledge, this is the first study that (1) systematically identifies both academic and practitioners' tools to be used in the comparison, (2) analyzes the code of the benchmarks to identify code characteristics that affect the performance of the tools and differentiate them, (3) conducts an in-depth analysis of the failures revealed by the tools, and (4) identifies concrete and specific future research directions.

As benchmark for this study, I used a set of 20 RESTful services selected among those used in related work and through a search on GitHub, focusing on Java/Kotlin open-source services that did not excessively rely on external resources. To select the tools for this evaluation, I performed a thorough literature search, which resulted in 8 academic and 11 practitioners' tools. Among those, I selected the tools that relied on commonly used REST API standards, such as OpenAPI, produced actual test cases, and were able to handle the benchmark of 20 services. The resulting set consisted of 10 tools overall: one white-box tool, EvoMasterWB [78, 45], and nine black-box tools, APiFuzzer [92], bBOXRT [85], Dredd [46], EvoMasterBB [93], RESTest [80], RESTler [9], RestTest-Gen [81, 82], Schemathesis [90], and Tcases [16]. In the chapter, I provide a characterization of these tools along several dimensions, including their underlying input-generation approach, their support for stateful API testing, and the types of test oracles they use.

I applied these tools to the benchmark of 20 services and evaluated their performance in terms of code coverage achieved (lines, branches, and methods exercised) and different kinds of fault-detection ability (generic errors, unique failure points, and unique library failure points). Through a thorough analysis of the results, I also investigated the strengths and weaknesses of the tools and of their underlying test-generation strategies.

Overall, all tools achieved relatively low line and branch coverage on many benchmarks, which indicates that there is considerable room for improvement. Two common limitations of many tools, in particular, involve the inability of (1) generating input values that satisfy specific constraints (e.g., parameters that must have a given format), and (2) sat-

Table 3.1: Overview of REST API testing techniques and tools.

Name	Website	Testing Approach	Test-Generation Technique	Stateful	Oracle	Parameter Generation	Version Used
EvoMasterWB	[95]	White-box	Evolutionary algorithm	Yes	Status code	Random, Mutation-based, and Dynamic	v1.3.0 [‡]
EvoMasterBB	[95]	Black-box	Random Testing	Yes	Status code	Random	v1.3.0 [‡]
REStler	[96]	Black-box	Dependency-based algorithm	Yes	Status code and predefined checkers	Dictionary-based and Dynamic	v8.3.0
REStGen	[82] [‡]	Black-box	Dependency-based algorithm	Yes	Status code and response validation	Mutation-based, Default, Example-based, Random, and Dynamic	v2.0.0
REStTest	[97]	Black-box	Model-based testing	No	Status code and response validation	Constraint-solving-based, Random, and Mutation-based	Commit 625b80e
Schemathesis	[98]	Black-box	Property-based testing	Yes	Status code and response validation	Random and Example-based	v3.12.3
Dredd	[46]	Black-box	Sample-value-based testing	No	Status code and response validation	Example-based, Enum-based, Default, and Dummy	v14.1.0
Tcases	[16]	Black-box	Model-based testing	No	Status code	Random or Example-based	v3.7.1
bBOXRT	[99]	Black-box	Robustness testing	No	Status code and behavioral analysis	Random and Mutation-based	Commit 7c894247
APIFuzzer	[92]	Black-box	Random-mutation-based testing	No	Status code	Random and Mutation-based	Commit e2b536f

[‡] I obtained this tool directly from its authors when it was a private tool, so the version is not in the website.

isfying dependencies among requests (e.g., this endpoint must be called before these other endpoints). In general, I found that accounting for dependencies among endpoints is key to performing effective REST API testing, but existing techniques either do not consider these dependencies or use weak heuristics to infer them, which limits their overall effectiveness.

This chapter also discusses lessons learned and implications for future research based on the results. For example, REST API testing techniques should leverage information embedded in the specification and the server logs to improve the quality of the test input parameters they generate. For another example, dependencies among services could be detected through static analysis, if the source code is available, and through natural language processing techniques applied to the service specification. Finally, in addition to discussing possible ways to improve REST testing approaches, I also present a proof-of-concept evaluation of these ideas that shows the feasibility of the suggested directions.

In summary, this chapter provides the following contributions:

- A comprehensive empirical study of automated test-generation tools for REST APIs that involves 10 academic and practitioners’ tools and assesses their performance on 20 benchmarks in terms of code coverage and different types of fault-detection ability.
- An analysis of the strengths and weaknesses of the tools considered and their underlying techniques, with suggestions for improvement and a discussion of the implications for future research.
- An artifact with the tools and benchmarks that can allow other researchers to replicate this work and build upon it [94].

3.2 Testing Tools Used In The Study

In this section, I first present the approach for selecting tools for the study and then describe each of the selected tools.

3.2.1 Tools Selection

For selecting tools for the study, I searched for REST-API-related papers published since 2011 in top venues in the areas of software engineering, security, and web technologies (e.g., ASE, CCS, FSE, ICSE, ISSTA, NDSS, S&P, TOSEM, TSE, USENIX, WWW, and WSDM).¹ I identified relevant papers via keyword search, using the terms “rest”, “api”, and “test” and by transitively following citations in the identified papers. Among the resulting set of papers, I selected those about techniques and tools that (1) rely on well-known REST API standards (i.e., OpenAPI, RAML, and API Blueprint), and (2) produce actual test cases.

This process resulted in 19 publicly-available tools: eight research tools—EvoMasterWB, EvoMasterBB, RESTler, RESTest, RestTestGen, bBOXRT, Schemathesis, and api-tester—and 11 practitioners’ tools—fuzz-lightyear, fuzzy-swagger, swagger-fuzzer, APiFuzzer, TnT-Fuzzer, RESTApiTester, Tcases, gadolinium, restFuzzer, Dredd, and kotlin-test-client.² More recently, I found a technical report [90] that describes two additional tools, cats [100] and got-swag [101]. Because Schemathesis, which is included in this study, outperforms them significantly [90], I did not include these tools in this comparison. I then eliminated tools that either did not work at all or failed to run on the benchmark of 20 services. It is worth noting that the remaining 10 tools, which are listed in Table 3.1, were also the most popular based on stars, watches, and forks in their GitHub repositories.

¹The latest search was performed in December 2021.

²I define as research tools those whose technical details are presented in research papers.

3.2.2 Collected Testing Tools

Table 3.1 presents an overview of the 10 tools I selected along several dimensions: the URL where the tool is available (column 2); whether the tool is white-box or black-box (column 3); test-generation strategy used by the tool (column 4); whether the tool produces test cases that exercise APIs in a stateful manner (column 5); the types of oracle used by the tool (column 6); the approach used by the tool to generate input parameter values (column 7); and the version of the tool (column 8).

EvoMaster [102] can test a REST API in either white-box or black-box mode. In the study, I used the tool in both modes. I refer to the tool in the black-box mode as *EvoMasterBB* and in the white-box mode as *EvoMasterWB*. Given a REST API and the OpenAPI specification, both tools begin by parsing the specification to obtain the information needed for testing each operation. *EvoMasterBB* performs random input generation: for each operation, it produces requests to test the operation with random values assigned to its input parameters. *EvoMasterWB* requires access to the source code of the API. It leverages an evolutionary algorithm (the MIO algorithm [103] by default) to produce test cases with the goal of maximizing code coverage. Specifically, for each target (uncovered) branch, it evolves a population of tests by generating new ones while removing those that are the least promising (i.e., have the lowest fitness value) for exercising the branch in each iteration until the branch is exercised or a time limit is reached. *EvoMasterWB* generates new tests through sampling or mutation. The former produces a test from scratch by either randomly choosing a number of operations and assigning random values to their input parameters (i.e., random sampling) or accounting for operation dependencies to produce stateful requests (i.e., smart sampling). The approach based on mutation, conversely, produces a new test by changing either the structure of an existing test or the request parameter values. *EvoMasterBB* and *EvoMasterWB* use an automated oracle that checks for service failures resulting in a 5xx status code. Recent extensions of the technique further improve the testing effectiveness by accounting for database states [104] and making better use of

resources and dependencies [105].

RESTler [9] is a black-box technique that produces stateful test cases to exercise “deep” states of the target service. To achieve this, RESTler first parses the input OpenAPI specification and infers producer-consumer dependencies between operations. It then uses a search-based algorithm to produce sequences of requests that conform to the inferred dependencies. Each time a request is appended to a sequence, RESTler executes the new sequence to check its validity. It leverages dynamic feedback from such execution to prune the search space (e.g., to avoid regenerating invalid sequences that were previously observed). An early version of RESTler relies on a predefined dictionary for input generation and targets finding 5xx failures. Newer versions of the technique adopt more intelligent fuzzing strategies for input generation [79] and use additional security-related checkers [83].

RestTestGen [81, 82] is another black-box technique that exploits the data dependencies between operations to produce test cases. First, to identify dependencies, RestTestGen matches names of input and output fields of different operations. It then leverages the inferred dependency information, as well as predefined priorities of HTTP methods, to compute a testing order of operations and produce tests. For each operation, RestTestGen produces two types of tests: nominal and error tests. RestTestGen produces nominal tests by assigning to the input parameters of an operation either (1) values dynamically obtained from previous responses (with high probability) or (2) values randomly generated, values provided as default, or example values (with low probability). It produces error tests by mutating nominal tests to make them invalid (e.g., by removing a required parameter). RestTestGen uses two types of oracles that check (1) whether the status code of a response is expected (e.g., 4xx for an error test) and (2) whether a response is syntactically compliant with the response schema defined in the API specification.

RESTest [80] is a model-based black-box input generation technique that accounts for inter-parameter dependencies [106]. An *inter-parameter dependency* specifies a con-

straint among parameters in an operation that must be satisfied to produce a valid request (e.g., if parameter A is used, parameter B should also be used). To produce test cases, RESTest takes as input an OpenAPI specification with the inter-parameter dependencies specified in a domain-specific language [107] and transforms such dependencies into constraints [108]. RESTest leverages constraint-solving and random input generation to produce nominal tests that satisfy the specified dependencies and may lead to a successful (2xx) response. It also produces faulty tests that either violate the dependencies or are not syntactically valid (e.g., they are missing a required parameter). RESTest uses different types of oracles that check (1) whether the status code is different from 5xx, (2) whether a response is compliant with its defined schema, and (3) whether expected status codes are obtained for different types of tests (nominal or faulty).

Schemathesis [90] is a black-box tool that performs property-based testing (using the Hypothesis library [109]). It performs negative testing and defines five types of oracles to determine whether the response is compliant with its defined schema based on status code, content type, headers, and body payload. By default, Schemathesis produces non-stateful requests with random values generated in various combinations and assigned to the input parameters. It can also leverage user-provided input parameter examples. If the API specification contains “link” values that specify operation dependencies, the tool can produce stateful tests as sequences of requests that follow these dependencies. Schemathesis also provides other features, such as concurrent test execution and storing and replaying requests.

Dredd [46] is another open-source black-box tool that validates responses based on status codes, headers, and body payloads (using the Gavel library [110]). For input generation, Dredd uses sample values provided in the specification (e.g., examples, default values, and enum values) and dummy values. Similar to Schemathesis, Dredd also supports the usage of hooks as custom functions that run before or after different testing steps.

Tcases [16] is a black-box tool that performs model-based testing. First, it takes as in-

put an OpenAPI specification and automatically constructs a model of the input space that contains key characteristics of the input parameters specified for each operation (e.g., a characteristic for an integer parameter may specify that its value is negative). Next, Tcases performs *each-choice* testing (i.e., 1-way combinatorial testing [111]) to produce test cases that ensure that a valid value is covered for each input characteristic. Alternatively, Tcases can also construct an example-based model by analyzing the samples provided in the specification and produce test cases using the sample data only. For test validation, Tcases checks the status code based on the request validity (as determined by the model).

bBOXRT [85] is a black-box tool that aims to detect robustness issues of REST APIs. Given a REST API and its OpenAPI specification, the tool produces two types of inputs, valid and invalid, for robustness testing. It first uses a random approach to find a set of valid inputs whose execution can result in a successful (i.e., 2xx) status code. Next, it produces invalid inputs by mutating the valid inputs based on a predefined set of mutation rules and exercises the REST API with those inputs. The approach involves manual effort to analyze the service behavior and (1) classify it based on a failure mode scale (the CRASH scale [112]) and (2) assign a set of behavior tags that provide diagnostic information.

APIFuzzer [92] is a black-box tool that performs fuzzing of REST APIs. Given a REST API and its specification, APIFuzzer first parses the specification to identify each operation and its properties. Then, it generates random requests conforming to the specification to test each operation and log its status code. Its input-generation process targets the body, query string, path parameter, and headers of requests and applies random generation and mutation to these values to obtain inputs. APIFuzzer uses the generated inputs to submit requests to the target API and produces test reports in the JUnit XML format.

3.3 Empirical Study

In this study, I investigated three research questions for the set of tools I considered and described in the previous section:

- **RQ1:** How much code coverage can the tools achieve?
- **RQ2:** How many error responses can the tools trigger?
- **RQ3:** What are the implications of the findings?

To answer RQ1, I evaluated the tools in terms of the line, branch, and method coverage they achieve to investigate their abilities in exercising various parts of the service code.

For RQ2, I compared the number of 500 errors triggered by the tools to investigate their fault-finding ability, as is commonly done in empirical evaluations of REST API testing techniques (e.g., [78, 9, 81, 10, 80, 84, 87, 90, 82, 14, 17]). I measured 500 errors in three ways. First, to avoid counting the same error twice, I grouped errors by their stack traces, and reported *unique 500 errors*. Therefore, unless otherwise noted, 500 errors will be used to denote unique 500 errors in the rest of the chapter. Second, different unique 500 errors can have the same failure point (i.e., the method-line pair at the top of the stack trace). Therefore, to gain insight into occurrences of such failure sources, I also measured *unique failure points*, which group stack traces by their top-most entries. Finally, I differentiated cases in which the unique failure point was a method from the web service itself from cases in which it was a third-party library used by the service; I refer to the latter as *unique library failure points*.

To answer RQ3, I (1) provide an analytical assessment of the studied tools in terms of their strengths and weaknesses and (2) discuss the implications of this study for the development of new techniques and tools in this space.

Next, I describe the benchmark of web services I considered (§subsection 3.3.1) and this experiment setup (§subsection 3.3.2). I then present the results for the three research questions (§subsection 3.3.3–subsection 3.3.5). I conclude this section with a discussion of potential threats to validity of the results (§subsection 3.3.6).

Table 3.2: RESTful web services used in the empirical study.

Name	Total LOC	Java/Kotlin LOC	# Operations
CWA-verification	6,625	3,911	5
ERC-20 RESTful service	1,683	1,211	13
Features-Service	1,646	1,492	18
Genome Nexus	37,276	22,143	23
Languagetool	677,521	113,277	2
Market	14,274	7,945	13
NCS	569	500	6
News	590	510	7
OCVN	59,577	28,093	192
Person Controller	1,386	601	12
Problem Controller	1,928	983	8
Project tracking system	88,634	3,769	67
ProxyPrint	6,263	6,037	117
RESTful web service study	3,544	715	68
Restcountries	32,494	1,619	22
SCS	634	586	11
Scout-API	31,523	7,669	49
Spring boot sample app	1,535	606	14
Spring-batch-rest	4,486	3,064	5
User management	5,108	2,878	23
Total	977,296	207,609	675

3.3.1 Web Services Benchmark

To create this evaluation benchmark, I first selected RESTful web services from existing benchmarks used in the evaluations of bBOXRT, EvoMaster, and RESTest. Among those, I focused on web services implemented in Java/Kotlin and available as open-source projects, so that I could use EvoMasterWB. This resulted in an initial set of 30 web services. I then searched for Java/Kotlin repositories on GitHub using tags “rest-api” and “restful-api” and keywords “REST service” and “RESTful service”, ranked them by number of stars received, selected the 120 top entries, and eliminated those that did not contain a RESTful specification. This additional search resulted in 49 additional web services. I tried installing and testing this total set of 79 web services locally and eliminated those that (1) did not have OpenAPI Specifications, (2) did not compile or crashed consistently, or (3) had a majority of operations that relied on external services with request limits (which would have unnaturally limited the performance of the tools). For some services, the execution

of the service through the EvoMaster driver class that I created crashed without a detailed error message. Because this issue was not affecting a large number of services, I decided to simply exclude the services affected.

In the end, this process yielded 20 web services, which are listed in Table 3.2. For each web service considered, the table lists its name, total size, size of the Java/Kotlin code, and number of operations. I obtained the API specifications for these services directly from their corresponding GitHub repositories.

3.3.2 Experiment Setup

I ran the experiments on Google Cloud e2-standard-4 machines running Ubuntu 20.04. Each machine had 4 2.2GHz Intel-Xeon processors and 16GB RAM. We installed on each machine the 10 testing tools from Table 3.1, the 20 web services listed in Table 3.2, and other software used by the tools and services, including OpenJDK 8, OpenJDK 11, Python 3.8, Node.js 10.19, and .NET framework 5.0. I also set up additional services used by the benchmarks, such as MySQL, MongoDB, and a private Ethereum network. This process was performed by creating an image with all the tools, services, and software installed and then instantiating the image on the cloud machines.

I ran each tool with the recommended configuration settings as described in the respective paper and/or user manual. Some of the tools required additional artifacts to be created (see also Section subsection 3.3.6). For RESTest, I created inter-parameter dependency specifications for the benchmark applications, and for EvoMasterWB, I implemented driver programs for instrumenting Java/Kotlin code and database injection. For lack of space, I omit here details of tool set up and configuration. A comprehensive description that enables the experiments and results to be replicated is available at the companion website [113].

I ran each tool for one hour, with 10 trials to account for randomness. Additionally, I ran each tool once for 24 hours, using a fresh machine for each run to avoid any interference between runs. For the one-hour runs, I collected coverage and error-detection data in 10-

Table 3.3: Average line, branch, and method coverage achieved and 500 errors found in one hour (E: unique error, UFP: unique failure point, ULFP: unique library failure point).

Tool	Line	Branch	Method	#500 (E/UFP/ULFP)
EvoMasterWB	52.76%	36.08%	52.86%	33.3 / 6.4 / 3.2
RESTler	35.44%	12.52%	40.03%	15.1 / 2.1 / 1.3
RestTestGen	40.86%	21.15%	42.31%	7.7 / 2 / 1
REStest	33.86%	18.26%	33.99%	7.2 / 1.9 / 1.1
bBOXRT	40.23%	22.20%	42.48%	9.5 / 2.1 / 1.3
Schemathesis	43.01%	25.29%	43.65%	14.2 / 2.8 / 2
Tcases	37.16%	16.29%	41.47%	18.5 / 3.5 / 2.1
Dredd	36.04%	13.80%	40.59%	6.9 / 1.5 / 0.9
EvoMasterBB	45.41%	28.21%	47.17%	16.4 / 3.3 / 1.8
APIFuzzer	32.19%	18.63%	33.77%	6.9 / 2.2 / 1.3

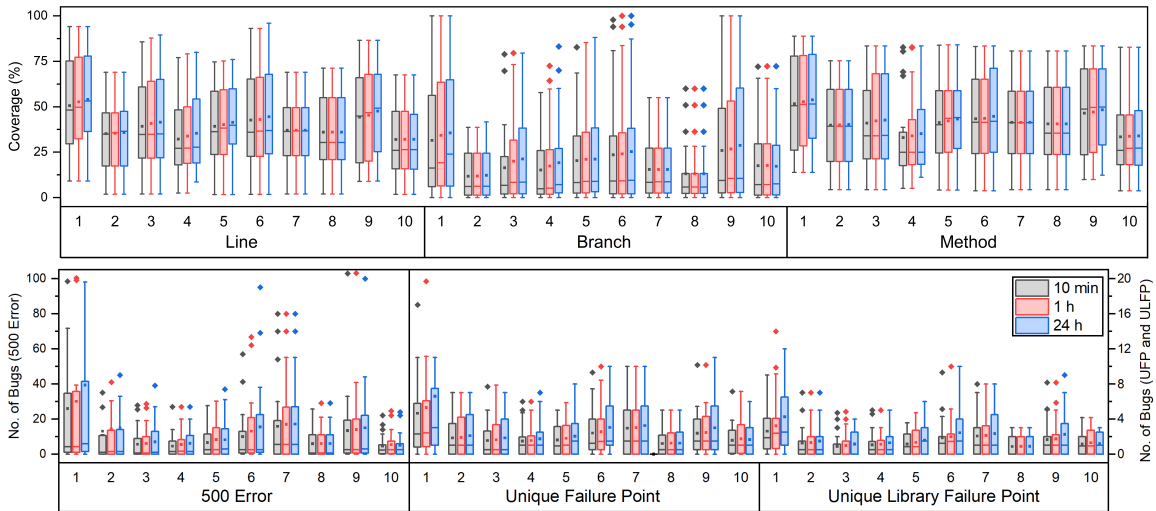


Figure 3.1: Code coverage achieved and number of unique 500 errors, unique failure points, and unique library failure points detected over all services by the ten tools in 10 minutes, 1 hour, and 24 hours (1: EvoMasterWB, 2: RESTler, 3: RestTestGen, 4: REStest, 5: bBOXRT , 6: Schemathesis, 7: Tcases, 8: Dredd, 9: EvoMasterBB, 10: APIFuzzer).

minute increments to investigate how these metrics increased over time. The data was used to solve RQ3. For the 24-hour run, I collected the data once at the end. I collected coverage information using the JaCoCo code-coverage library [114].

3.3.3 RQ1: Coverage Achieved

Table 3.3 presents the results for line, branch, and method coverage achieved, and 500 errors, unique failure points, and unique library failure points detected by the 10 tools in

one hour. The reported results are the average of the coverage achieved and errors found over all services and all ten trials. Unlike Table 3.3, which shows only average results, Figure 3.1 presents coverage data (top) and error-detection results (bottom) as box plots. In the figure, the horizontal axis represents the 10 tools, and the vertical axis represents, for each tool, the results achieved by that tool after 10 minutes, 1 hour, and 24 hours. Each box represents average, median, standard deviation, min, and max for 20 data points, one per service considered. For the 10-minute and 1-hour runs, each point represents the coverage achieved (or number of errors revealed) averaged over 10 trials. For the 24-hour runs, the values reported are from single trials.

Overall coverage achieved Table 3.3 shows that the tools did not achieve a high coverage in general. In one hour, the best-performing tool (EvoMasterWB) achieved less than 53% line coverage, less than 37% branch coverage, and less than 53% method coverage. The other black-box tools achieved lower coverage; the best-performing tool among them is EvoMasterBB, with 45.41% line coverage, 28.21% branch coverage, and 47.17% method coverage. I identified three key factors responsible for these low coverage rates.

- **Parameters value generation.** The tools generate many invalid requests that are rejected by the services and fail to exercise service functionality in depth. In most cases, this occurs when the parameters take domain-specific values or have data-format restrictions. As an example, for a parameter that takes email IDs as values, one of the services enforced the condition that the email should contain a ‘@’ character. Tools wasted a significant amount of time attempting to pass that check, but they all failed in the one-hour and 24-hour runs in this experiment.
- **Operations dependency detection.** Identifying producer-consumer dependencies between operations is key to generating stateful tests. Existing tools either do not account for such dependencies or use a simple heuristic approach, which can lead to many false positives and false negatives in the dependencies computed. Failing to

identify a dependency between producer A and consumer B can result in inadequate testing of B due to not testing A first. This is because B may require the execution of A to set the service state needed for its testing (e.g., by adding valid data to the database) or obtain valid parameter values. As an example, for Languagetool, most of the tools fail to identify a dependency between the two operations `POST /check` (consumer) and `GET /languages` (producer) in 24 hours. As a result, they did not leverage the output of the producer, which contained language information, and were unable to produce valid requests for the consumer.

- **Mismatch between APIs and their specifications.** The tools produce test cases by leveraging the API specifications, which are expected to faithfully reflect the API implementations and define, for each endpoint, supported HTTP methods, required input parameters, responses, and so on. Unfortunately, this is not always the case and, if the specification does not match the API implementation, the tools can produce incomplete, invalid, or even no requests at all for exercising the affected operations.

Existing tools fail to achieve high code coverage due to limitations of the approaches they use for generating parameter values and detecting operation dependencies. The effectiveness of the tools is also hindered by mismatches between API implementations and specifications.

Coverage increase over time As I mentioned above, I collected coverage data in 10-minute increments, as well as for 24 hours. Figure 3.1 illustrates how the coverage rate increased from 10 minutes to 1 hour, and to 24 hours (more detailed results are available in the artifact [113]). As the figure shows, in many cases, the tools already achieved their highest level of code coverage in 10 minutes. However, there were several cases in which code coverage kept increasing over time. I investigated the 6 services for which the tools manifested this behavior—Features-Service, NCS, Restcountries, SCS, RESTful Web Ser-

vice Study, and User management—and found that they all have some distinct characteristics (compared to the other 14 services). Specifically these services generally have simpler input parameters that do not take domain-specific or otherwise constrained values, so additional input generation tends to cover additional code. Conversely, for the services that have domain-specific or constrained parameter values (e.g., “email should contain @” or “year should be between 1900 and 2200”), the tools tend to hit a coverage wall relatively soon because they are inherently unable to generate inputs that satisfy these requirements.

The coverage achieved by testing tools grows over time on services that have simpler input parameters with no domain-specific or constrained values.

3.3.4 RQ2: Error Responses Triggered

The ultimate goal of testing is to find bugs. In this section, I focus on comparing the testing tools in terms of their fault-finding ability, which I measured in terms of the numbers of unique 500 errors, failure points, and library failure points detected. Column 5 in the previously presented Table 3.3 provides this information averaged over the services considered and the trials performed. As the table shows, EvoMasterWB is the best performer by a wide margin, followed by Tcases, EvoMasterBB, RESTler, and Schemathesis. The box plot at the bottom of Figure 3.1 presents a more detailed view of these results by illustrating the distribution of errors detected across services and the increase in errors detected over time. The “500 Error” segment of the plot shows that EvoMasterWB outperforms all the other tools in terms of the median values as well, although with a larger spread.

EvoMasterWB, by having access to source code and performing coverage-driven testing, achieves significantly higher coverage than black-box tools. However, there are cases in which EvoMasterWB cannot produce requests covering some service functionality, whereas black-box tools can.

The figure also shows that the number of unique failure points is considerably smaller than the number of unique errors—on average, there are 3 to 7 times fewer failure points than errors, indicating that several 500 errors occur at the same program points but along different call chains to those points. Another observation is that approximately half of the unique failure points occur in library methods. A more detailed analysis of these cases revealed that failure points in library methods could have more serious consequences on the functionality and robustness of a service, in some cases also leaving it vulnerable to security attacks. The reason is that failures in service code mostly originate at statements that throw exceptions after performing some checks on input values; the following fragment is an illustrative example, in which the thrown exception is automatically mapped to a 500 response code by the REST API framework being used:

```
if (product == null) {throw new ObjectNotFoundException(name);}
```

Conversely, in the case of library failure points, the root cause of the failures was often an unchecked parameter value, possibly erroneous, being passed to a library method, which could cause severe issues. I found a particularly relevant example in the ERC-20 RESTful service, which uses an Ethereum [115] network. An Ethereum transaction requires a fee, which is referred to as gas. If an invalid request, or a request with insufficient gas, is sent to Ethereum by the service, the transaction is canceled, but the gas fee is not returned. This is apparently a well-known attack scenario in Blockchain [116]. In this case, the lack of suitable checks in the ERC-20 service for requests sent to Ethereum could have costly repercussions for the user. I found other examples of unchecked values passed from the service under test to libraries, resulting in database connection failures and parsing errors.

The severity of different 500 errors can vary considerably. Failure points in the service code usually occur at throw statements following checks on parameter values. Failure points outside the service, however, often involve erroneous requests that have been accepted and processed, which can lead to severe failures.

I further investigated which factors influence the error-detection ability of the tools. First, I studied how the three types of coverage reported in Table 3.3 correlate with the numbers of the 500 errors found using the Pearson's Correlation Coefficient [117]. The results showed that there is a strong positive correlation between the coverage and number of 500 errors (coefficient score is ~ 0.7881 in all cases). Although expected, this result confirms that it makes sense for tools to use coverage as a goal for input generation: if a tool achieves higher coverage, it will likely trigger more failures.

There is a strong positive correlation between code coverage and number of faults exposed. Tools that achieve higher coverage usually also trigger more failures.

Second, I looked for patterns, not related to code coverage, that can increase the likelihood of failures being triggered. This investigation revealed that exercising operations with different combinations of parameters can be particularly effective in triggering service failures. The failures in such cases occur because the service lacks suitable responses for some parameter combinations. In fact, Tcases and Schemathesis apply this strategy to their advantage and outperform the other tools. Generating requests with different input types also seems to help in revealing more faults.

Exercising operations with various parameter combinations and various input types helps revealing more faults in the services under test.

3.3.5 RQ3: Implications of the Results

I next discuss the implications of the results for future research on REST API testing and provide an analytical assessment of testing strategies employed by the white-box and black-box testing tools I considered.

```

1 public static String subject(String directory , String file) {
2   int result = 0;
3   String[] fileparts = null;
4   int lastpart = 0;
5   String suffix = null;
6   fileparts = file.split(".");
7   lastpart = fileparts.length - 1;
8   if (lastpart > 0) { ... } //target branch
9   return "" + result;
10 }

```

Figure 3.2: A method used for parsing the file suffix in the SCS web service.

Implications for Techniques and Tools Development

Better input parameter generation These results show that the tools I considered failed to achieve high code coverage and could be considerably improved. Based on these findings, one promising direction in this context is better input parameter generation.

In particular, for white-box testing, analysis of source code can provide critical guidance on input parameter generation. In fact, EvoMasterWB, by performing its coverage-driven evolutionary approach, achieves higher coverage and finds more 500 errors than any of the black-box tools. There are, however, situations in which EvoMasterWB’s approach cannot direct its input search toward better coverage. Specifically, this happens when the fitness function used is ineffective in computing a good search gradient. As an example, for the code shown in Figure 3.2, which parses the suffix of a file, EvoMasterWB always provides a string input `file` that leads to `lastpart` in line 8 evaluating to 0. The problem is that EvoMasterWB cannot derive a gradient from the condition `lastpart > 0` to guide the generation of inputs that exercise the branch in line 8. In this case, symbolic execution [118, 119] could help find a good value of `file` by symbolically interpreting the method and solving the constraint derived at line 8.

For black-box testing approaches, which cannot leverage information in the source code, using more sophisticated testing techniques (e.g., combinatorial testing at 2-way or higher interaction levels) could be a promising direction [14, 12, 120]. Also, black-box testing tools could try to leverage useful information embedded in the specification and

other resources to guide input parameter generation.

The analysis in Section subsection 3.3.5 below shows that using sample values for input parameter generation can indeed lead to better tests. Therefore, another possible way to improve test generation would be to automatically extract sample values from parameter descriptions in the specification. For example, the description of the input parameter `language`, shown in the following OpenAPI fragment for Languagetool (lines 4–9), suggests useful input values, such as “en-US” and “en-GB”:

```
1 "name": "language",
2 "in": "formData",
3 "type": "string",
4 "description": "A language code like 'en-US', 'de-DE', 'fr', or 'auto' to guess the
5 language automatically (see 'preferredVariants' below). For languages with variants
6 (English, German, Portuguese) spell checking will only be activated when you specify
7 the variant, e.g. 'en-GB' instead of just 'en'." , "required": true
```

Another source of useful hints for input generation can be response messages from the server, such as the following one:

```
1 Received: "HTTP/1.1 400 Bad Request\r\nDate: Wed, 28 Apr 2021
2 00:38:32 GMT\r\nContent-length: 41\r\n\r\nError: Missing 'text'
3 or 'data' parameter"
```

To investigate the feasibility of leveraging input parameter descriptions provided in the REST API specifications to obtain valid input values, I implemented a proof-of-concept text-processing tool that checks whether there are suggested values supplied in the parameter description for a given input parameter. I applied the tool to two of the services, OCVN and Languagetool, for which the existing testing tools failed to obtain high coverage (less than 10% line code coverage for most cases). With the help of the tool, I identified developer-suggested values for 934 (32%) of the 2,923 input parameters. This preliminary result suggests that leveraging parameter descriptions for input generation may be a feasible and promising direction.

Along similar lines, I believe that natural-language processing (NLP) [121] could be leveraged to analyze the parameter description and extract useful information. For example,

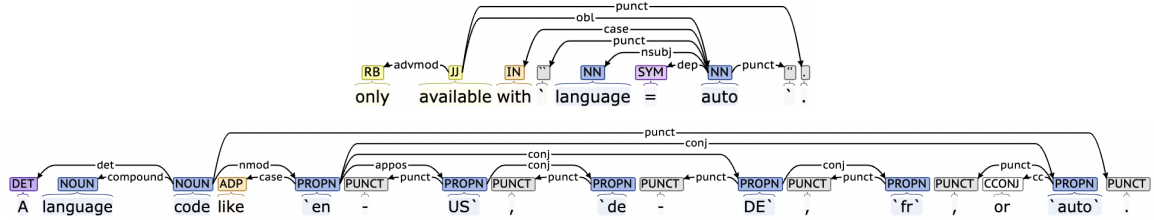


Figure 3.3: Parsed dependency graphs of `preferredVariants` parameter description (top) and `language` parameter description (bottom) from `Languagetool`'s OpenAPI specification.

a technique may first perform token matching [122] to identify what parameters from the specification are mentioned in server messages and then use parts-of-speech tagging [123] and dependency parsing [124] to infer parameter values.

Also in this case, to investigate the feasibility of this approach, I implemented a proof-of-concept prototype that parses dependency information from natural-language descriptions in OpenAPI specifications and server messages and collects nouns, pronouns, conjunctions, and their dependencies. The prototype detects parameter names with simple string matching on nouns and pronouns, and relationships between parameters via conjunctions and dependencies. As an example, the top part of Figure 3.3 shows the parsed dependencies for parameter `preferredVariants` of endpoint `/check`. By analyzing the tokens `with`, `language`, and `auto` with the connected dependencies (case, punct, nsbj, and dep), the prototype can determine that parameter `language` must be set to “auto”. This simple approach was able to automatically detect 8 of the 12 unique inter-parameter dependencies that I manually found in the benchmark APIs. It was also able to detect useful parameter values. For example, none of the black-box tools studied could generate values for parameter `language`; yet, the prototype detected useful values such as “en-US”, “de-DE”, and “fr” (the parsed dependencies are shown in the bottom part of Figure 3.3). As before, these preliminary results show the feasibility of applying NLP techniques to descriptions in OpenAPI specifications and server messages for improving REST API testing.

Better support for stateful testing As discussed in Section subsection 3.3.3 and also stressed in related work [102, 9], producing stateful tests by inferring producer-consumer relationships between operations is key to effective REST API testing. Current tools, such as RestTestGen and RESTler, rely on simple heuristics to infer the producer-consumer dependencies and are inaccurate in identifying such dependencies, which can easily lead to false positives and negatives. There is a need for more sophisticated approaches that address the flexibility issues that I discuss later in Section subsection 3.3.5. These approaches should be able to account for responses dynamically generated and to match fields of different types while still being precise (e.g., by not identifying a dependency between two consumers that rely on the same parameters to execute).

To determine whether two operations have a dependency relationship, one possible approach would be to check whether they have “related” properties, such as input parameters and response fields, that have the same or similar names. To explore the feasibility of this direction, I performed a quick case study in which I manually identified each operation that uses path parameters and has a dependency relationship with at least another operation. For each of them, I then identified its path parameter(s) and compared its textual similarity with other operations’ response fields using the NLTK’s scoring library [125]. In this way, by considering the top-3 matches for each parameter, I were able to correctly identify almost 80% of the operations involved in a dependency relationship. Based on the findings in RQ1, I expect that a similar approach could considerably help tools to achieve higher code coverage.

I also believe that machine learning [126] could help identify producer-consumer relationships and related dependencies. For example, one could train, via supervised learning [127], a classifier that accounts for a variety of features related to operation dependencies (e.g., HTTP methods, shared input parameters, field matching on dynamic object, field types) and then use this classifier to make predictions on potential dependencies.

```

1 if ("male".equals(sex)) {
2     if ("mr".equals(title) || "dr".equals(title) ||
3         "sir".equals(title) || "rev".equals(title) ||
4         "rthon".equals(title) || "prof".equals(title)) {
5         result = 1;
6     }
7 } else if ("female".equals(sex)) {
8     if ("mrs".equals(title) || "miss".equals(title) ||
9         "ms".equals(title) || "dr".equals(title) ||
10        "lady".equals(title) || "rev".equals(title) ||
11        "rthon".equals(title) || "prof".equals(title)) {
12        result = 0;
13 }}

```

Figure 3.4: Sample code from the SCS web service.

Analytical Comparison of White-Box and Black-Box Tools

Next, I present an analytical assessment of the tools with respect to strengths and weaknesses of their approaches for generating input parameter values and sequences of API requests. I also provide illustrative examples taken from the benchmark services.

White-box vs. black-box tools Among the tools I considered, EvoMasterWB is the only one that performs white-box testing. By having access to the API source code and performing coverage-driven testing, EvoMasterWB achieves higher coverage than the other tools—according to Table 3.3, it achieves ~53% line coverage, ~36% branch coverage, and ~53% method coverage.

To illustrate, Figure 3.4 shows an if-statement in which the two branches (lines 5 and 12) are exercised only by test cases produced by EvoMasterWB. The if-statement is responsible for handling requests for operation `GET /api/title/{sex}/{title}` of the SCS web service. To cover these branches, a tool must produce valid requests with relevant string values for parameters `sex` and `title`. Using an evolutionary algorithm with a fitness function that measures branch distance, EvoMasterWB successfully generates values “male” for `sex` and “dr” for `title` to exercise line 5, and values “female” and “prof” for those parameters to cover line 12. The black-box tools are unable to do this by using random and/or sample values.

```

1 public List<String>
2   getConfigurationsNamesForProduct(String productName) {
3   List<String> configurationsForProduct=new ArrayList<String>();
4   for (ProductConfiguration productConfiguration :
5     productsConfigurationsDAO.findByProductName(productName)) {
6     configurationsForProduct.add(productConfiguration.getName());
7   }
8   return configurationsForProduct;
9 }

```

Figure 3.5: A method of Features-Service used to find configuration names associated with a product.

Another benefit of EvoMasterWB’s testing strategy is that once it produces a test case exercising a branch A, it will not generate similar test cases to exercise A again. This is due to its MIO algorithm [103], which handles test case generation for the target branches separately and focuses on uncovered branches.

However, there are also cases in which EvoMasterWB fails to create a sequence of requests for covering some API functionality, whereas a black-box tool is able to. Consider the code fragment shown in Figure 3.5, which handles requests for operation O_2 : GET /products/{productName}/configurations of Features-Service. To cover line 6, a request must specify a product that exists and has configurations associated with it. None of the requests created by EvoMasterWB satisfy both conditions: operation O_1 : POST /products/{productName}/configurations/{configurationName} must be called before O_2 to associate configurations with a product, and EvoMasterWB fails to recognize this producer-consumer relation. In contrast, Schemathesis uses a testing strategy that orders O_1 before O_2 and leverages sample values from the API specification to link two operations with the same input values. It can therefore generate a sequence that creates a product, adds configurations to it, and retrieves the configurations—a sequence that covers line 6.

Assessment of black-box tools Although black-box tools seems to be less effective than EvoMasterWB in terms of coverage achieved, which can also often result in fewer faults triggered, they have wider applicability by virtue of being agnostic to the language used for

the service implementation. Among the black-box tools, EvoMasterBB and Schemathesis achieve better coverage than the other tools in terms of all the three metrics considered (unique 500 errors, failure points, and library failure points).

EvoMasterBB uses an evolutionary algorithm to generate successful requests, create sequences from them, and discover in this way operation dependencies. This way of operating allows it to find more valid request sequences than other tools that just use a randomized approach.

Schemathesis is the next best black-box tools. One characteristic of this tool is that it considers example values provided in the API specification, which lets it leverage the domain knowledge encoded in such examples. Moreover, it reuses values in creating request sequences, which enables it to successfully create covering sequences such as that for the loop body in Figure 3.5.

RestTestGen also has similar features, but I empirically found that its heuristic algorithm, which relies upon the matching of response fields and parameter names to infer producer-consumer dependencies between operations, yields many false positives and false negatives. This issue weakens the effectiveness of its stateful testing and leads to slightly lower code coverage than EvoMasterBB.

RESTler tries to infer producer-consumer relationships by leveraging feedback from processed requests. An important limiting factor for RESTler, however, is that it relies on a small set of dictionary values for input generation, which hinders its ability to exercise a variety of service behaviors.

Dredd does not perform random input generation but uses dummy values and input examples (provided in the specification) to produce requests. This prevents Dredd from generating invalid requests exercising uncommon behaviors in a service.

Finally, the other tools' random-based approaches are unlikely to produce valid test cases needed for input mutation, which limits their overall effectiveness.

Overall, black-box tools fail to achieve high coverage because they largely rely on ran-

dom testing and/or leverage a limited set of sample data for input generation. Among these tools, Schemathesis and EvoMasterBB performed better than the others in the experiments due to some specific characteristics of the input generation approaches they use.

3.3.6 Threats to Validity

Like any empirical evaluation, the study in this chapter could suffer from issues related to internal and external validity. To mitigate threats to internal validity, I used the implementations of the testing tools provided by their authors (the tool versions used are listed in Table 3.1). The implementation consists of Python code for analyzing the log files of the web services to compute unique 500 errors, failure points, and library failure points. I thoroughly tested and spot checked the code and manually checked the testing results (coverage and faults) for a sample of the tools and web services to gain confidence in the validity of the results.

As for the threats to external validity, because this evaluation is based on a benchmark of 20 RESTful services, the results may not generalize. I note, nevertheless, that the benchmark includes a diverse set of services, including services that have been used in prior evaluations of REST API testing techniques.

Another potential threat is that I ran the services locally, which may result in different behavior than a real-world deployment. For example, I set up a private Ethereum network instead of using the Ethereum main network for the ERC-20 service. However, I believe that this is unlikely to affect the results in any significant way. Furthermore, I manually checked that the locally-installed services behaved as expected, including for the ERC-20 deployment.

In this empirical study, I used the OpenAPI specifications provided with each web service in the benchmark. I observed that some specifications were incomplete, which may limit the tools' ability to achieve higher coverage or identify more bugs. To explore the extent of this issue, I analyzed over 1,000 specifications from APIs.guru and checked

them using Schemathesis, a tool that detects inconsistencies between API specifications and server behavior. I discovered that over 99% of the specifications contained mismatches, mostly due to missing parts of the response schema. This widespread issue arises because the OpenAPI Specification grammar does not mandate the inclusion of a complete response schema, leading many specifications to omit this crucial information.

The response schema is especially important for building effective stateful testing or specification coverage-guided testing in REST API testing tools, as it defines expected response structures, allowing tools to validate outputs and create meaningful sequences of API calls. Without this information, tools may struggle to generate comprehensive test cases or accurately detect certain types of bugs. Interestingly, this limitation impacted some tools in this study more than others. For instance, tools that rely heavily on response validation, such as Tcases, RESTler, EvoMaster, Schemathesis, and RestTestGen, showed reduced effectiveness when working with incomplete specifications. In contrast, simpler fuzzers that do not consider stateful testing, like Dredd, bBOXRT, RESTest, and APIFuzzer, were less affected by the missing response schema. In light of these findings, it's clear that incomplete specifications are a common issue in real-world scenarios. Therefore, the specifications used in this study, despite their limitations, are representative of what testing tools would typically encounter in practice. This highlights the importance of developing testing tools that can perform effectively even with incomplete API documentation.

RESTest requires additional information over what is in OpenAPI specifications. Specifically, it requires inter-parameter dependencies information [106]. I therefore studied the parameters of each API endpoint for the web services in the benchmarks and added such dependencies (if any) as annotations to the OpenAPI specifications. Although it was simple to create such annotations, and it typically took me only a few minutes per specification, the quality of the specifications might have affected the performance of the tool. Unfortunately, this is an unavoidable threat when using RESTest.

Finally, EvoMasterWB requires, for each service under test, a test driver that performs various operations (e.g., starting and stopping the web service). Building such driver can require a non-trivial amount of manual effort. For the 10 web services I selected from previous work, I obtained the existing drivers created by the EvoMasterWB author. For the remaining 10 web services, however, I had to create a driver myself. Also in this case, this was a fairly trivial task, which I were able to complete in just a few minutes for each web service. And also in this case, the only way to avoid this threat would be to exclude EvoMasterWB from the set of tools considered, which would considerably decrease the value of this comparative study.

3.4 Conclusion

To gain insights into the effectiveness of existing REST API testing techniques and tools, I performed an empirical study in which I applied 10 state-of-the-art techniques to 20 RESTful web services and compared them in terms of code coverage achieved and unique failures triggered. I presented the results of the study, along with an analysis of the strengths and weaknesses of the techniques, summarized the lessons learned, and discussed implications for future research. The experiment infrastructure, data, and results are publicly available [113].

CHAPTER 4

ENHANCING REST API TESTING WITH NLP TECHNIQUES

4.1 Introduction

OpenAPI specifications are for the most part machine-readable, but they also let developers add, in description fields, natural-language descriptions of some aspects of an API. In fact, many OpenAPI specifications rely heavily on natural-language comments to describe additional details of operation parameters.

Although many automated testing tools have been presented in the literature that leverage the *machine-readable* part of OpenAPI specifications to guide test generation, most of these tools completely ignore the *human-readable* descriptions in these specifications. I believe that this is a major limitation of the state of the art in REST API testing, as these natural-language descriptions can contain relevant and useful information for testing.

Consider, for instance, the `description` field associated with the parameter `language` in lines 21–22 of Figure 4.1(a), which provides examples of valid values (i.e., language codes) that the parameter can take. This information can be leveraged by a testing tool to create requests with valid parameter values. Conversely, testing tools that ignore this information often generate requests with random values for `language`, which is likely to result in trivial failures. Figure 4.1(b) shows two other forms of useful information in parameter descriptions: *parameter constraints* and *parameter formats*. For example, the description in line 15 specifies the constraint that parameter `count` must have a maximum value of 50, while the description for `Accept-Language` (lines 9–10) states that values for that parameter must be a comma-delimited list. This additional information can clearly help testing tools generate more effective tests.

Natural-language descriptions in OpenAPI specifications can also help identify *inter-*

```
1 paths:
2   /check:
3     post:
4       summary: Check a text
5       description: >- The main feature - check a text with LanguageTool
6                   for possible style and grammar issues.
7     parameters:
8       - name: text
9         in: formData
10        type: string
11        description: The text to be checked. This or 'data' is required.
12        required: false
13       - name: data
14         in: formData
15         type: string
16         description: The text to be checked.
17         required: false
18       - name: language
19         in: formData
20         type: string
21         description: >- A language code like 'en-US', 'de-DE', 'fr'
22                     or 'auto'.
23         required: true
```

(a) LanguageTool [128] OpenAPI specification excerpt

```
1 paths:
2   /search:
3     get:
4       operationId: Web_Search
5       parameters:
6         - name: Accept-Language
7           in: header
8           type: string
9           description: A comma-delimited list of one or more languages
10                      to use for user interface strings.
11         - name: count
12           in: query
13           type: integer
14           format: int32
15           description: The maximum value is 50.
```

(b) Bing Web Search OpenAPI specification excerpt

Figure 4.1: Sample OpenAPI specification fragments to illustrate rule extraction from human-readable descriptions.

parameter dependencies [106]. Consider line 11 in Figure 4.1(a), which states that either the `text` or `data` parameter must be specified when invoking endpoint `/check`.

To the best of my knowledge, no existing technique can automatically extract *all* these types of information from descriptions in OpenAPI specifications in a flexible way and leverage it to improve testing. To bridge this gap, I present NLP2REST, a fully automated technique that (1) leverages specially tailored natural language processing (NLP) techniques to infer different types of rules from the human-readable part of OpenAPI specifications, (2) encodes these rules using OpenAPI-compliant keywords, and (3) adds the encoded rules to the original specification to produce an enhanced specification that can be transparently utilized by any REST API test generator.

The set of rules extracted through NLP techniques can contain spurious or incorrect rules due to the ambiguity of natural-language descriptions, limitations of NLP-based rule inference, or mismatch between API specification and implementation (schema-mismatch faults, for example, are common [129]). To mitigate this issue, NLP2REST includes a validation step that checks the NLP-extracted rules against the API implementation by crafting and executing validation test cases and discards rules that fail such validation.

To evaluate this approach, I performed multiple studies using a set of nine REST services, including popular services such as Spotify and the FDIC Bank Data API. External developers manually computed the ground truth by independently reviewing the OpenAPI specifications and identifying all the OpenAPI rules present. The developers then compared the rules extracted by NLP2REST against this ground truth. NLP2REST achieved $\sim 50\%$ precision and $\sim 94\%$ recall in the purely NLP-based rules extraction; after validation, the precision increased to $\sim 79\%$, with a 3% reduction in recall.

Second, I compared this approach with RestCT [130]. To the best of my knowledge, RestCT is the only other approach in the literature that uses NLP to extract rules from the human-readable part of an OpenAPI specification, but it only supports inter-parameter dependencies. This approach was considerably more effective, as it extracted 15 of the 19

inter-parameter dependencies in the benchmark considered, whereas RestCT was unable to extract any.

Finally, I evaluated whether the enhanced specifications created by NLP2REST can improve the performance of existing REST API testing tools. I studied the performance of eight state-of-the-art REST test generation tools, when provided with the original and enhanced specifications, in terms of (1) code coverage, (2) rate of successful requests, rejected requests, and server error responses, and (3) unique faults found. Using the enhanced specification, the coverage achieved by the testing tools increased, on average, from 11.35% to 23.10% for branch coverage, from 24.96% to 37.52% for statement coverage, and from 22.13% to 33.54% for method coverage. The enhanced specifications contributed to increasing the rate of successful requests (+20%) and server error responses (+2%) and decreasing the rate of rejected requests (-7%). The number of unique server error responses returned by the API increased, on average, by 4%, with a maximum increase of 98.9%. I believe these results are promising, especially considering that there are several directions in which the technique can be extended and improved.

The main contributions of the work in this chapter are:

- A novel technique for extracting rules from natural-language descriptions in OpenAPI specifications, validating the rules to improve their accuracy, and generating enhanced OpenAPI specifications that existing REST API testing tools can transparently use. It is worth noting that the enhanced OpenAPI specifications may also be useful in other contexts, such as to provide developers with additional documentation.
- Empirical results showing that NLP2REST can extract rules accurately, and the extracted rules can considerably improve the performance of existing REST API testing tools.
- An artifact [131] with the NLP2REST tool and experiment data that can be used for replicating and extending this work.

Table 4.1: Types of rules identified from natural-language descriptions in API specifications in the preliminary study.

REST service	API endpoint(s)	No. of Params	Rule Categories			
			Param type/fmt	Param cons	Param ex	Oper cons
Bing Web Search	Search	21	15	10	14	9
Forte	Create application	99	13	67	41	8
Foursquare	Search venues	14	2	0	0	3
GitHub	Get user repos	5	1	0	3	2
Google Geocoding	Geocode request	9	4	0	4	5
Google Maps	Nearby search	11	0	1	2	4
PayPal	Create invoice	114	12	43	5	2
Stripe	Create coupon	28	5	9	3	9
	Create product					
Tumblr	Create post	35	10	2	10	1
	Get blog likes					
Yelp	Search businesses	14	0	0	9	4
Total		350	62	132	91	47

4.2 Preliminary Study

To assess the kind of information available in the human-readable part of OpenAPI specifications, I manually analyzed the specifications of a set of API endpoints from a benchmark used in previous work [106], which includes 10 real-world REST APIs. This analysis led to the identification of four categories of rules.

Parameter type/format. Rules that define the type and format of a parameter (e.g., `type: string` and `format: date`).

Parameter constraint. Rules that restrict the possible values of a parameter (e.g., a parameter’s maximum value).

Parameter example. Rules that specify sample values that a parameter can take.

Operation constraint. Rules that define some specific conditions parameters must satisfy (e.g., inter-parameter dependencies).

Table 4.1 lists, for each of the OpenAPI specifications and endpoints I analyzed, the number of parameters, the number of rules I identified in the natural-language descriptions, and their categories. In total, I identified over 330 rules, consisting of 62 parameter types/formats, 131 parameter constraints, 91 parameter examples, and 47 operation con-

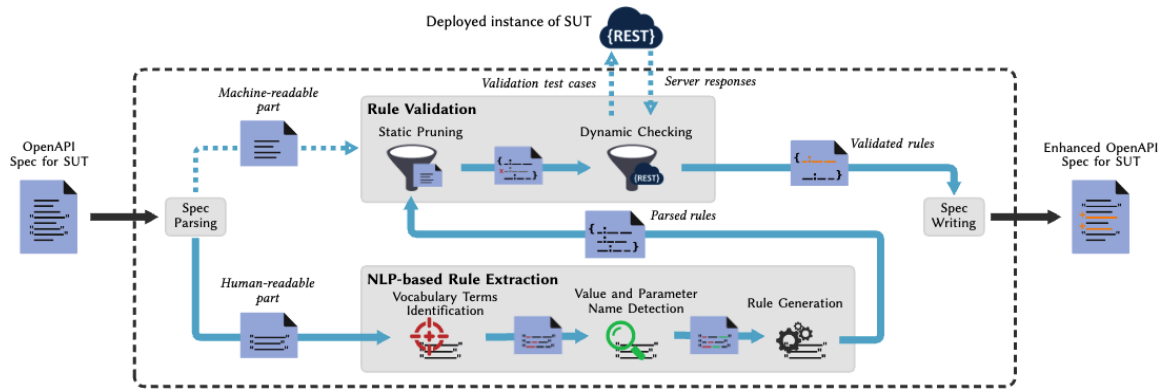


Figure 4.2: Overview of NLP2REST approach.

straints. These data show that there is considerable scope for identifying meaningful rules from natural-language descriptions in OpenAPI specifications, and guided the development of this approach.

4.3 Approach

In this section, I first provide an overview of the whole NLP2REST approach (§subsection 4.3.1), and then describe its two main components in detail: NLP-based Rule Extraction (§subsection 4.3.3) and Rule Validation (§subsection 4.3.4).

4.3.1 Approach Overview

Figure 4.2 presents an overview of this technique. The inputs to the technique are the OpenAPI specification of the service under test (SUT) and a deployed instance of the SUT. The output of the technique is an enhanced version of the specification, in which rules extracted from natural-language descriptions have been added using OpenAPI-supported keywords (i.e., part of the core OpenAPI syntax or OpenAPI extensions [132]). For instance, the rules extracted from the descriptions in lines 21–22 of Figure 4.1(a) and line 15 of Figure 4.1(b) would be added using core OpenAPI syntax “`examples: {1:en-US, 2:de-DE, 3:fr, 4:auto}`” and “`maximum: 50`”, respectively. Conversely, the inter-parameter dependency rule inferred from the descrip-

tion in line 11 of Figure 4.1(a) would be added using the OpenAPI extension keyword “`OnlyOne(text, data)`” [133]. (For this example, as I discuss later, the NLP-based analysis of NLP2REST extracts the rule as *inclusive or*, i.e., `Or(text, data)`, and the validation step modifies it to *exclusive or*, i.e., `OnlyOne(text, data)` based on the actual service implementation).

Initially, NLP2REST parses the OpenAPI specification to extract its human-readable and machine-readable parts, where the former consists of natural-language text contained in `description` fields associated with parameters in the specification. After this preliminary step, NLP2REST takes these two parts as input and performs *NLP-based Rule Extraction* and *Rule Validation*. Specifically, the NLP-based Rule Extraction module analyzes the human-readable part of a specification to find useful information to extract as rules that could be added to the machine-readable part of the specification, whereas the Rule Validation module validates the extracted rules and discards rules that fail such validation.

The NLP-based Rule Extraction phase looks for potential rules by scanning for a set of search terms using a custom Word2Vec [134] model, pre-trained on OpenAPI terminology (*Vocabulary Terms Identification*). If a description contains one of the search terms, it next looks for values and inter-parameter dependencies using a constituents grammar (*Value and Parameter Name Detection*). Finally, this phase creates a rule for each (1) keyword and corresponding value and (2) inter-parameter dependency detected. The rules are in the form of key-value pairs (e.g., “`maximum: 50`”) or inter-parameter constraints (e.g., “`Or(text, data)`”, “`IF videoDef THEN type==video`”).

The Rule Validation phase checks the candidate rules generated by the previous phase against the service specification and implementation. It (1) statically analyzes combinations of rules to discard incompatible ones (*Static Pruning*) and (2) generates validation test cases (i.e., HTTP requests) for various rule combinations and executes them against the deployed SUT. When a test case is successfully executed (i.e., a 2XX status code is

obtained), the corresponding rules are further processed by a fine-tuning phase. Rules that pass the complete validation process are marked as *valid*.

The final step of NLP2REST (*Spec Writing*) adds the validated rules, in machine-readable format, to the original specification, yielding an enhanced OpenAPI specification for the SUT.

4.3.2 Terminology

OpenAPI specifications describe a RESTful API in terms of its *operations* (i.e., endpoints and HTTP verbs) and their corresponding input and output parameters (names, types, and possibly formats). I use the term *OpenAPI vocabulary* to indicate the set of keywords and string literals used in OpenAPI specifications, where *string literals* correspond to predetermined values for a given *keyword*, such as the value `string` for keyword `type`. I refer to an entry in the OpenAPI vocabulary as an *OpenAPI vocabulary term*. A complete list of such terms can be found in the OpenAPI specification standard [135]. The term *rule* indicates either a key-value pair, for rules that involve a keyword and possible values for that keyword, or an inter-parameter constraint, for rules that involve inter-parameter dependencies. In turn, an *inter-parameter constraint* can have one of three formats: `IF condition THEN constraint`, `Operator(parameter1, ..., parametern)`, or a combination of the two (e.g., `IF condition THEN Operator(parameter1, ..., parametern), where Operator is one of four inter-parameter dependency types Or, OnlyOne, AllOrNone, and ZeroOrOne [133]. This approach currently supports extraction of simpler forms of these rules where the Operator arguments are simple parameters; in the more general formulation of these operators, an argument can be a predicate [133].`

Algorithm 1 Search term detection

```
1: procedure SEARCHTERMDETECTION(s, vocabularyTerms, sentencesTermsMap)
2:   similarityThreshold  $\leftarrow$  0.7 ▷ Threshold for cosine similarity
3:   for vt in vocabularyTerms do ▷ loop1
4:     searchTerms  $\leftarrow$  getSearchTerms(vt)
5:     for w in s do ▷ loop2
6:       for st in searchTerms do ▷ loop3
7:         if COSINESIMILARITY(w, st)  $\geq$  similarityThreshold then
8:           sentencesTermsMap.add(s, vt)
9:         continue loop1
10:        end if
11:      end for
12:    end for
13:  end for
14: end procedure
```

4.3.3 NLP-based Rule Extraction

Extracting rules from description fields is challenging due to the many ways in which concepts can be expressed in natural language. For instance, the fact that the maximum value of a parameter is 50 can be stated as “the maximum value is 50”, “the value is up to 50”, “the value can’t be larger than 50”, and so on. Similarly, inter-parameter dependencies can be described in many ways, such as “you must also set X” or “X must also be specified” [106]. Therefore, simple pattern-based approaches tend to be ineffective, as this evaluation comparing NLP2REST against one such technique [14] shows (§subsection 4.4.3). I instead propose a more flexible approach that leverages a custom pre-trained NLP model and constituency-parse-tree-based sentence analysis [136] and consists of three main parts: vocabulary terms identification (§subsubsection 4.3.3), value and parameter name detection (§subsubsection 4.3.3), and rule generation (§subsubsection 4.3.3).

Vocabulary Terms Identification

This step analyzes the description field of each parameter to identify sentences that may contain rules. To define this part of the technique, I first identified common linguistic patterns used to describe rules and involving OpenAPI vocabulary terms based on the preliminary study in §section 4.2 and the common patterns for inter-parameter dependencies

described in [106]. From these linguistic patterns, I then identified *search terms* that, if present, may indicate that the sentence contains the corresponding rule or OpenAPI vocabulary term. This resulted in a total of 55 mappings. Although I do not show the mappings here for space reasons, they are available in the artifact [131]. I would like to note that, although these relations were derived from the manual analysis of the REST services in Table 4.1, this evaluation was performed on a different set of services to avoid overfitting and provide evidence of the generalizability of the results.

The use of a fixed set of search terms is limited because different words with similar meanings could be used instead of a specific term. To address this problem, I use word embeddings [134] to detect semantic similarity between words. Although many pre-trained models are publicly available, they are trained using general data sources and may not be ideal in the specific domain of OpenAPI documents. I therefore trained a custom Word2Vec model, *restW2V*, on parameter descriptions taken from numerous OpenAPI documents. Specifically, I trained the model on 1,875,607 text sets taken from 4,064 REST API specifications, using FastText [137], so as to create a custom model more suited to the terminology and phrases commonly used in OpenAPI specifications.

Given the mappings I identified and *restW2V*, NLP2REST processes, using Algorithm Algorithm 1, each sentence in each description of each parameter in the OpenAPI specification of the SUT. The algorithm takes as input a sentence s , a set of vocabulary terms, and a map (*sentencesTermsMap*) that associates sentences with the vocabulary terms. In this context, the set of vocabulary terms used is the complete set of OpenAPI vocabulary terms. The outer loop (`loop1`) iterates over all the possible OpenAPI vocabulary terms. For each vocabulary term vt , it first extracts the relevant search terms associated with that term (line 3) and then iterates over each word w in s (`loop2`) and search term st (`loop3`). If the cosine similarity (according to *restW2V*) of w and st is 0.7 or greater (line 7), the algorithm adds s and vt to the map provided as input (line 8) and continues with the next vocabulary term (line 9). When all the sentences have been processed by

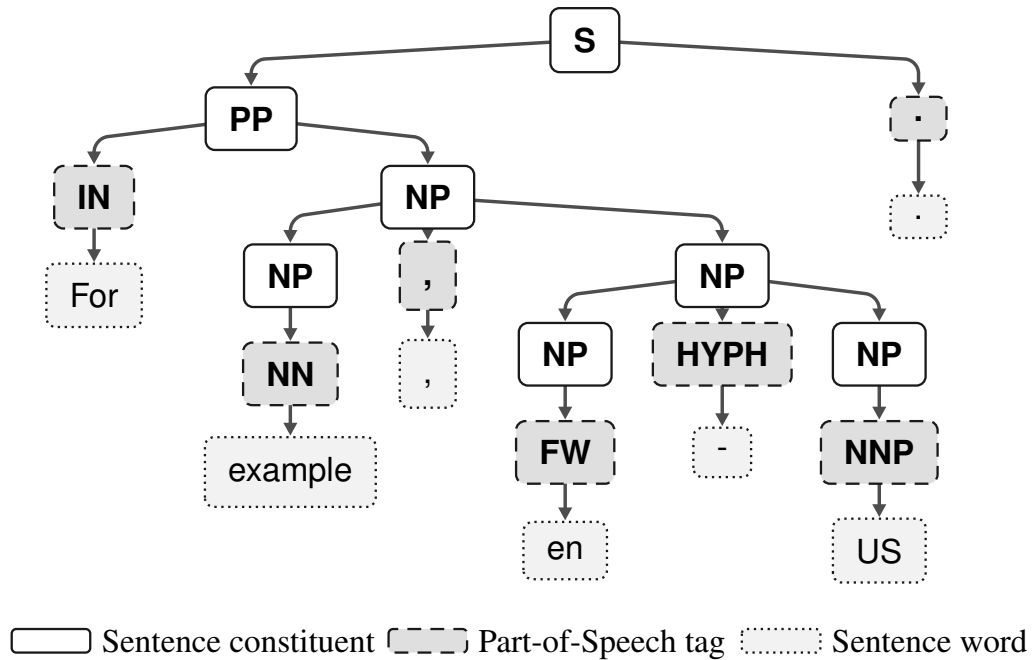


Figure 4.3: An example of a constituency parse tree.

the algorithm, *sentencesTermsMap* contains a map from each sentence to any OpenAPI vocabulary term related to that sentence.

Intuitively, this step identifies all the sentences that describe something related to an OpenAPI vocabulary term, such as a keyword, and that may therefore contain a rule. For instance, for the sentence “A language code like ‘en-US’, ‘de-DE’, ‘fr’ or ‘auto’” from Figure 4.1(a), Algorithm Algorithm 1 would detect the search terms `like`, which is associated with the OpenAPI vocabulary term `example` [131], and would therefore map that sentence to such term. In the next section (§subsection 4.3.3), I describe how NLP2REST would then identify, in that same sentence, values `en-US`, `de-DE`, `fr` and `auto`. These values, together with keyword `example` would define the possible rule “`examples: {1:en-US, 2:de-DE, 3:fr, 4:auto}`”.

Value and Parameter Name Detection

The NLP-based Rule Extraction module employs two strategies to detect the values associated with the previously identified OpenAPI vocabulary terms in a sentence. First, it uses

Algorithm 2 Value detection

```
1: procedure VALUEDETECTION(sentence)
2:   values  $\leftarrow$   $\emptyset$ 
3:   stopwords  $\leftarrow$  GetStopwords() ▷ Load NLTK based stopwords
4:   regexValues  $\leftarrow$  RegularExpDetection(sentence) ▷ First try detection using regular expressions
5:   if regexValues =  $\emptyset$  then ▷ If no values detected using regular expressions
6:     tree  $\leftarrow$  sentence.getConstituencyParseTree()
7:     constituents  $\leftarrow$  tree.getConstituents(NP)
8:     pos  $\leftarrow$  tree.getPoS(NOUN, NUM)
9:     for node in constituents do
10:      leafNodes  $\leftarrow$  node.getLeaves()
11:      childPos  $\leftarrow$  [child.getPoS()forchildinnode.getChildren()]
12:      if "," is in childPos then
13:        commaSets  $\leftarrow$  node.getCommaSeparatedSets()
14:        for set in commaSets do
15:          if set  $\neq$  'and' and set  $\neq$  'or' then
16:            values  $\leftarrow$  values  $\cup$  set
17:          end if
18:        end for
19:      else
20:        for leaf in leafNodes do
21:          if leaf.getParent()  $\in$  pos then
22:            values  $\leftarrow$  values  $\cup$  leaf
23:          end if
24:        end for
25:      end if
26:    end for
27:  else
28:    values  $\leftarrow$  regexValues ▷ If values were detected using regular expressions, use them
29:  end if
30:  values  $\leftarrow$  values – stopwords ▷ Filter out stopwords
31:  return values
32: end procedure
```

regular expressions to parse enumerated or quoted strings. If this approach does not yield results, it leverages the *constituency parse tree* [138] as a fallback method.

The *constituency parse tree* represents the grammatical structure of a sentence in tree format. This method utilizes part-of-speech (PoS) tagging and sentence constituents for its operation. *PoS tagging* categorizes each word in a sentence into its grammatical function (e.g., noun, verb, adjective, adverb), while *sentence constituents* refer to the structural components of a sentence, such as phrases or clauses. For instance, Figure 4.3 illustrates the constituency parse tree for the sentence “For example, en-US.”, a simplified version of a sentence in Figure 4.1(a). Each terminal node in the tree corresponds to a word in

the sentence, its parent node is a PoS tag, and its other ancestor nodes correspond to its constituent nodes.

This preliminary work suggests that enumerated or quoted strings frequently contain critical values in OpenAPI descriptions. Hence, I introduced a regular expression-based detector to directly extract such values. Currently, the detector recognizes enumerated strings starting with ”- ” or ”* ”, and quoted strings using double quotes (”), single quotes (’), backticks (`), along with bold strings wrapped in asterisks (*). I intend to further enhance the regular expression detection capabilities to accommodate more markdown features in future work.

NLP2REST employs Algorithm 2 to detect values in a sentence via regular-expression and constituency-parse-tree analyses. The algorithm first initializes a set of stopwords, which are derived from NLTK [139] and supplemented with OpenAPI-specific vocabulary terms (line 3). It then attempts to detect values using regular expressions (line 4) and saves them if successful (line 28). Conversely, if no values are detected using regular expressions, the algorithm analyzes the constituency parse tree for the sentence to identify all the NP (noun phrase) constituents (lines 6–7). A noun phrase is a group of words that function as a single unit within a sentence and include a noun. If a noun phrase contains commas, the algorithm assumes that these commas separate different sets of words, each of which could potentially represent a value. It then extracts these comma-separated sets and considers each set as a potential value (lines 13–18). If a comma is not present within the noun phrase, the algorithm identifies all the words whose PoS tag corresponds to a noun or a number as potential values (lines 20–24). The algorithm then filters out any stopwords from the detected values (line 30) and returns the remaining values as the final result (line 31).

In addition to identifying relevant values, NLP2REST also needs to pinpoint parameter names within sentences, which is crucial for detecting dependencies between different parameters. To accomplish this task, NLP2REST employs the Inter-Parameter Dependency

Algorithm 3 Inter-parameter dependency parser

```
1: procedure IPDPARSER(sentence, operation)
2:   tree  $\leftarrow$  sentence.getConstituencyParseTree()
3:   sbar_node  $\leftarrow$  tree.getConstituents(SBAR)  $\triangleright$  Subtree node representing a subordinate clause
4:   main_node  $\leftarrow$  tree.getConstituents(S) - sbar_node  $\triangleright$  Subtree node representing a sentence without
   subordinate clauses
5:   main  $\leftarrow$  main_node.getLeavesAsSentence()
6:   sbar  $\leftarrow$  sbar_node.getLeavesAsSentence()
7:   main_voc  $\leftarrow$   $\emptyset$ , main_param  $\leftarrow$   $\emptyset$ , main_val  $\leftarrow$   $\emptyset$ 
8:   sbar_voc  $\leftarrow$   $\emptyset$ , sbar_param  $\leftarrow$   $\emptyset$ , sbar_val  $\leftarrow$   $\emptyset$ 
9:   IPDTerms  $\leftarrow$  subset of OpenAPI vocabulary terms relevant for IPDs
10:  sentencesTermsMap  $\leftarrow$   $\emptyset$ 
11:  SEARCHTERMDETECTION(main, IPDTerms, sentencesTermsMap)
12:  main_voc  $\leftarrow$  sentencesTermsMap[main]
13:  sentencesTermsMap  $\leftarrow$   $\emptyset$ 
14:  SEARCHTERMDETECTION(sbar, IPDTerms, sentencesTermsMap)
15:  sbar_voc  $\leftarrow$  sentencesTermsMap[sbar]
16:  for pm in operation.getParameters do
17:    for word in main do
18:      if COSINESIMILARITY(word, pm)  $\geq$  0.7 then
19:        main_param  $\leftarrow$  main_param  $\cup$  pm
20:      end if
21:    end for
22:    for word in sbar do
23:      if COSINESIMILARITY(word, pm)  $\geq$  0.7 then
24:        sbar_param  $\leftarrow$  sbar_param  $\cup$  pm
25:      end if
26:    end for
27:  end for
28:  ExampleTerms  $\leftarrow$  relevant terms for Parameter example (see §section 4.2)
29:  sentencesTermsMap  $\leftarrow$   $\emptyset$ 
30:  SEARCHTERMDETECTION(main, ExampleTerms, sentencesTermsMap)
31:  if sentencesTermsMap[main]  $\neq$   $\emptyset$  then  $\triangleright$  There is a match
32:    main_val  $\leftarrow$  VALUEDETECTION (main)
33:  end if
34:  sentencesTermsMap  $\leftarrow$   $\emptyset$ 
35:  SEARCHTERMDETECTION(sbar, ExampleTerms, sentencesTermsMap)
36:  if sentencesTermsMap[sbar]  $\neq$   $\emptyset$  then  $\triangleright$  There is a match
37:    sbar_val  $\leftarrow$  VALUEDETECTION (sbar)
38:  end if
39:  return (main_voc, main_param, main_val, sbar_voc, sbar_param, sbar_val)
40: end procedure
```

Parser (IPDParser), as outlined in Algorithm Algorithm 3. This algorithm takes as input a sentence and its corresponding operation (i.e., the operation that contains the parameter described within that sentence). Utilizing Algorithms Algorithm 1 and Algorithm 2, IPDParser extracts inter-parameter constraints, parameter names, and values from the main and subordinate clauses of the sentence. To do so, it first generates a constituency parse tree for the given sentence and extracts the main subordinate clauses from this tree (lines 2–6), which forms the foundation for the extraction process. (Note that, for simplicity, the current version of IPDParser in Algorithm Algorithm 3 does not handle multiple main or subordinate clauses.)

Next, IPDParser populates the sets of relevant vocabulary terms, parameter names, and values from both the main and subordinate clauses. It selects a set of OpenAPI vocabulary terms that are pertinent to inter-parameter dependencies [106] and applies the `SearchTermDetection` algorithm to identify these terms within the main and subordinate clauses (lines 9–15). The algorithm then enters a loop (lines 16–27), in which it inspects each parameter name from the input operation, searching for corresponding terms in the main and subordinate clauses. This matching process uses cosine similarity based on `restW2V`, similarly to Algorithm Algorithm 1. Subsequently, the algorithm selects a set of terms that are relevant for parameter example rules (line 28, also see §section 4.2). It again leverages the `SearchTermDetection` algorithm to identify these terms within the main (lines 29–30) and subordinate (lines 34–35) clauses. When a match is found, it utilizes the `ValueDetection` algorithm to pinpoint corresponding values for the main (line 32) and subordinate (line 37) clauses.

Finally, IPDParser returns a tuple that encapsulates the identified relevant terms, parameters, and values for both the main and subordinate clauses (line 39), and which is used to identify inter-parameter dependencies within the given sentence.

Rule Generation

The final step of NLP-based Rule Extraction involves generating OpenAPI-compliant rules based on the information computed in the previous steps. Rule generation is performed in a syntax-driven manner for each vocabulary term identified in each description. For example, consider the description in lines 21–22 of Figure 4.1(a). The search term detection step would match the word “like” with the vocabulary term `examples` [131], and the value-detection step would discover the values `en-US`, `de-DE`, `fr`, and `auto`. This information would then be combined in the format conforming to the syntax of the `examples` keyword: “`examples: {1:en-US, 2:de-DE, 3:fr, 4:auto}`”.

For inter-parameter dependencies, IPDParser identifies inter-parameter dependency terms, parameter names, and values for both the subordinate and main clauses, as described above. Rules would then be generated for each clause, separately, again in a syntax-driven way. For example, for the description in line 11 of Figure 4.1(a), the vocabulary term “or” and parameter names “text” and “data” would be identified in the main clause, from which a rule would be created as `Or(text, data)`. In the presence of a subordinate clause, and inter-parameter dependencies within it, rules would be generated for both the subordinate and the main clauses and would then be concatenated using the `Requires` dependency.

4.3.4 Rule Validation

The NLP-based rule extractor may generate spurious or incorrect rules due to various factors, including ambiguity in natural-language descriptions, limitations of NLP-based rule inference, or mismatches between the API specification and its implementation. To address these issues, the NLP2REST framework includes a Rule Validation step, which is essential for ensuring the accuracy and relevance of the derived rules.

Rule Validation consists of a *static-pruning phase*, where incompatible rule combinations are discarded through static analysis, and a *dynamic-checking phase*, where rule combinations are validated dynamically. This process involves submitting requests to a

deployed API instance and analyzing the HTTP responses. It is inherently combinatorial, requiring the evaluation of 2^n combinations of rules in the worst-case scenario, where n is the number of rules to validate. Each API operation is examined in detail, considering all associated rules, both for individual parameters and inter-parameter dependencies.

To enhance the practical applicability of this methodology, my colleague Davide Corradini, the author of RestTestGen [8], defined the tasks for the validation process. Subsequently, I developed a Python script tailored for validating rules within the specific context of the benchmark services. This script performs targeted validations that confirm the efficacy of the rule extraction and pruning methods in controlled environments. Additionally, to generalize the validation process across diverse API scenarios, Davide utilized RestTestGen, his mature REST API testing tool. This extension allowed me to conduct a more comprehensive evaluation of the NLP2REST framework's capabilities by testing a broader range of services beyond the initial benchmark.

Static Pruning

Given a set of rules associated with an API operation, the static-pruning phase discards syntactically incompatible rule combinations. These are combinations where at least one rule conflicts with the OpenAPI specification. For example, if the OpenAPI specification indicates that a parameter is of type number, but a rule suggests a value of type string, the rule is considered incompatible and discarded.

To enforce this, Davide Corradini defined a set of rule compatibility policies derived from the OpenAPI standard and its extensions. If all the rules within a combination comply with these policies, I consider the combination syntactically compatible. Due to space constraints, only a subset of these policies is provided here; the full set is available in the artifact [131].

- If a parameter is required (`required: true`), the same parameter cannot appear in the inter-parameter dependency rules `Or`, `OnlyOne`, `ZeroOrOne`, or `AllOrNone`, as

these rules would imply that the parameter is not actually required.

- A `maximum: a` rule is allowed only for numeric parameters and only if, for the parameter in question, there are no `minimum: b` rules such that $b > a$.
- Rule `exclusiveMaximum: true` is only allowed if the parameter is numeric and a maximum value is defined for the parameter.
- Example, enum, and default values must match the type defined for the parameter.

When an incompatibility in a combination is identified, this step discards all other combinations that contain the same incompatible rules. This pruning reduces the number of combinations to check, significantly improving the overall efficiency of the static pruning process.

Dynamic Checking

Dynamic checking tries to verify the validity of rule combinations by interacting with a deployed instance of the REST API. In general, each rule can be:

- correct and necessary to drive test generation to a successful request (e.g., a correct `enum` value);
- incorrect without causing the request to fail (e.g., a required parameter that is not actually required in the implementation);
- incorrect causing the request to fail (e.g., a wrong `enum` value).

Using different combinations of rules, starting from the largest one, will eventually allow the technique to identify a maximal combination that (1) includes all the rules necessary to generate a valid request (i.e., 2XX response code), and (2) does not contain any of the rules that cause the request to fail (4XX response code). Note that this step currently

does not consider server error responses (5XX) as successful for rule validation purposes— in this experience, a server error might occur during the processing of a request and hide a subsequent 4XX response that would occur had the processing of the request been completed. It is worth noting, however, that there is a tension between eliminating these cases and possibly discarding valid rule combinations that may lead to the identification of actual failures in the services under test. For each combination of rules, this approach generates a request that considers all of its rules, along with the rules in the original specification. When a request is valid, NLP2REST classifies the rules in the current combination as *potentially valid*, indicating that the combination may contain both correct rules and incorrect rules that do not cause the request to fail. Rules are further checked, and possibly fixed, in the subsequent fine-tuning phase.

Fine tuning Fine tuning is meant to further assess the validity of rules. Consider, for instance, an API operation that accepts a non-mandatory parameter `p` and an incorrectly extracted `required: true` rule for that parameter. This rule is semantically incorrect, but would not cause the request to fail with a 4XX return code. Therefore, the rule would not be discarded during the above described dynamic checking and would be considered as potentially valid.

Fine tuning implements a strategy that replays the successful requests identified in the previous phase while applying a series of mutations, so as to further validate the each potentially valid rule. Different types of rules are validated with different strategies, and fine-tuning provides a precise validation strategy for 22 out of the 26 rule types supported by NLP-based Rule Extraction. For example, to validate rules of type `required`, the request is replayed twice, with and without the required parameter. If the request without the parameter is also successful, it is discarded. For another example, to validate `if-then` rules, the request is replayed twice, once applying the predicate and once applying its negation. If the former request is successful and the latter is rejected, the rule is confirmed

as valid.

In addition to validating rules, fine tuning can also attempt to repair inter-parameter dependency rules `Or`, `OnlyOne`, `AllOrNone`, and `ZeroOrOne`. Consider, for instance, the `Or(text, data)` rule extracted from the specification of `LanguageTool` in Figure 4.1(a). During fine tuning, the successful request obtained in the previous step is mutated and replayed four times: (1) with both `text` and `data` parameters; (2) with the `text` parameter only; (3) with the `data` parameter only; and (4) with neither `text` nor `data`. Based on the status codes of the responses obtained in the four cases, fine tuning computes a new rule that better complies with the API implementation. In this particular case, the new rule would be `OnlyOne(text, data)` because requests (2) and (3) are accepted by the API, while requests (1) and (4) result in errors. In particular, the rejection of request (1) indicates that the two parameters cannot be used simultaneously, contrary to what an `Or` rule states.

In summary, this approach considers as *validated* those rules that belong to a combination not discarded by static pruning, for which dynamic checking could generate a successful request, and that could be successfully processed by fine tuning. These rules are included in the enhanced OpenAPI specification.

4.4 Evaluation

In this section, I present an empirical evaluation of NLP2REST. Specifically, I assess the effectiveness of NLP-based Rule Extraction and Rule Validation, compare this approach with RestCT [130] (a related tool that performs pattern-matching-based rule extraction from OpenAPI specifications), and investigate how enhanced specifications generated by NLP2REST can help in improving the performance of state-of-the-art REST API testing tools.

4.4.1 Research Questions

I formulated the following research questions for the evaluation:

Table 4.2: Effectiveness of NLP-based Rule Extraction and Rule Validation.

REST Service	No. of Rules in Ground Truth	NLP-based Rule Extraction						Rule Validation					
		TP	FP	FN	Precision	Recall	F ₁	TP	FP	FN	Precision	Recall	F ₁
FDIC	45	42	36	3	54%	93%	68%	42	25	3	63% (+16%)	93% (-)	75% (+10%)
Genome Nexus	81	79	3	2	96%	98%	97%	79	3	2	96% (-)	98% (-)	97% (-)
LanguageTool	20	20	12	0	63%	100%	77%	18	2	2	90% (+44%)	90% (-10%)	90% (+17%)
OCVN	17	15	2	2	88%	88%	88%	13	1	4	93% (+5%)	76% (-13%)	84% (-5%)
OhSome	14	13	66	1	16%	93%	28%	12	11	2	52% (+217%)	80% (-14%)	63% (+126%)
OMDb	2	2	0	0	100%	100%	100%	2	0	0	100% (-)	100% (-)	100% (-)
REST Countries	32	28	1	4	97%	88%	92%	28	0	4	100% (+4%)	88% (-)	93% (+2%)
Spotify	88	83	68	5	55%	94%	69%	82	28	6	75% (+36%)	93% (-1%)	83% (+19%)
YouTube Mock	34	30	126	4	19%	88%	32%	28	9	6	76% (+294%)	82% (-7%)	79% (+150%)
Total	333	312	314	21	50%	94%	65%	304	79	29	79% (+58%)	91% (-3%)	85% (+31%)

RQ1: How effective is NLP-based Rule Extraction in inferring rules from natural-language descriptions in OpenAPI specifications?

RQ2: How effective is Rule Validation in pruning incorrect rules?

RQ3: How does NLP2REST compare with related NLP-based rule-extraction tools?

RQ4: Can the enhanced specification generated by NLP2REST improve the performance of REST API testing tools?

To answer RQ1, I compared the rules inferred by NLP-based Rule Extraction against a manually created ground truth and computed true positives, false positives, false negatives, precision, recall, and F₁ score. The ground truth was created by four graduate students with expertise in REST API specifications. These students were not privy to the rules identified in this preliminary experiment (§section 4.2). Instead, I provided them with documentation of the OpenAPI syntax and the OpenAPI specifications, and instructed them to infer rules from descriptions. I did not impose any other restrictions on their interpretation. The students worked independently and compared their results, resolving any discrepancies through discussion until they reached a consensus.

To answer RQ2, I compared the validated rules against the ground truth, computing the same metrics that I used for RQ1.

To answer RQ3, I compared NLP2REST with RestCT [130], a test generation tool that uses pattern matching to extract inter-parameter dependencies from text descriptions.

Specifically, I compared the tools in terms of the number of correct rules extracted.

To answer RQ4, I compared the performance of state-of-the-art REST API testing tools in two settings: (1) the tools take as input the original OpenAPI specification, and (2) the tools take as input the enhanced OpenAPI specification generated by NLP2REST. To measure tool performance, I used line, branch, and method coverage achieved, and the frequency of successful (2XX), client error (4XX), and server error (5XX) status codes. I expected the enhanced specifications generated by NLP2REST to help the testing tools generate a higher number of successful requests (2XX status codes) and a higher number of requests that trigger server errors (5XX status codes), while generating fewer invalid requests (4XX status codes). Consequently, I expected the testing tools to achieve higher code-coverage and fault-detection rates (the latter indicated by the number of server errors triggered).

4.4.2 Experiment Setup

REST API Benchmark

To build the evaluation benchmark, I started with the services used in two recent empirical studies in the area: the first one evaluating online testing of REST APIs [140], and the second one comparing automated REST API testing tools [141]. To the best of my knowledge, the first study includes the largest number of commercial REST APIs in the literature, whereas the second one contains the largest number of open-source REST APIs, with 33 REST services in total.

I put this set of services through a filtering process. The goal was to focus on industrial-sized services, as current testing tools are already very effective on small services but tend to struggle with larger ones [141]. Following the approach of Martin-Lopez, Segura, and Ruiz-Cortés [140], I consider a service to be industrial-sized if its source code consists of more than 10,000 lines of code (LoC). Therefore, I excluded from the benchmark services with less than 10k LoC. I also excluded services that I had used to build the NLP model

for NLP2REST, so as to avoid potential overfitting. This filtering resulted in a benchmark of 12 services. I then tested each service using each REST API testing tool considered (§subsection 4.4.2) for one hour to check for any execution problems. Unfortunately, I found that one service, Amadeus v2, had an authentication issue. I contacted the Amadeus developers, who informed me that they were in the process of fixing the issue; however, the fix was not available at the time of this experimentation. Additionally, three services—DHL, Marvel, and YouTube—implement rate limiting (i.e., they limited the number of allowed requests per hour), which caused more than half of the requests in this testing sessions to be blocked. I therefore had to exclude DHL and Marvel and replaced YouTube with YouTube Mock (a locally-hosted version of the YouTube service included in one of the benchmark I considered [140]).

After this process, the final benchmark consisted of nine REST services: Federal Deposit Insurance Corporation (FDIC), LanguageTool, OhSome, Open Movie Database (OMDb), REST Countries, Genome Nexus, OCVN, Spotify, and YouTube Mock.

REST API Testing Tools

To select the tools for this study, I evaluated state-of-the-art black-box testing tools for REST APIs that generate tests based on OpenAPI specifications. Specifically, I first selected the top seven performers from the tools I studied in the previous work [141]. I then considered two additional tools, Morest [142] and RestCT [130], which were developed recently. However, I had to exclude RestCT [130] due to compatibility issues with most of the APIs in the benchmark. (I notified the developers of RestCT about these issues; unfortunately, they were still in the process of addressing them at the time of this experimentation, so I could not include the tool.) The final set of eight tools includes EvoMasterBB [45], bBOXRT [143], Morest [142], ResTest [11], RESTler [9], RestTestGen [8], Schemathesis [13], and Tcases [16].

Experiment Procedure

I ran the experiments using the cloud computing service provided by Google Cloud. In particular, I used ten e2-standard-4 machines running Ubuntu 20.04. Each machine had 24 2.2GHz Intel-Xeon processors and 128 GB of RAM.

I ran the testing tools with a time budget of one hour. This choice was based on the experience in previous work [141], which showed that the code coverage achieved by fuzzers tends to plateau within one hour. To account for randomness, I repeated the experiments 10 times and collected average metrics across the 10 runs, resulting in 10 hours of execution per testing tool for each service.

To collect code coverage information, I used JaCoCo [114], which allowed me to instrument the service code. I could measure coverage for only the four open-source services in the benchmark (Genome Nexus, LanguageTool, OCVN, and YouTube Mock); for the remaining services, source code is unavailable for instrumentation.

4.4.3 Experiment Results

RQ1: Effectiveness of NLP-based Rule Extraction

Table 4.2 presents the results for RQ1. In the table, Column 2 shows the number of rules in the ground truth for each service. Columns 3–8 present data on the accuracy of NLP-based Rule Extraction: the number of rules extracted correctly (true positives or TP), the number of erroneous rules extracted (false positives or FP), the number of rules missed (false negatives or FN), and the aggregate metrics (precision, recall, and F_1 score).

The results demonstrate that NLP-based Rule Extraction is effective in extracting rules from the human-readable part of OpenAPI specifications, with an overall recall of 94% (i.e., missing only 21 of the 338 rules in the ground truth). This high recall rate is fairly consistent across services, ranging from a minimum of 88% (for YouTube Mock) to a maximum of 100% (for LanguageTool and OMDb).

The precision results were not as good, with approximately half of the 626 extracted rules being false positives (i.e., 50% precision on average, across services). However, for four of the services (Genome Nexus, OCVN, OMDb, and REST Countries), the technique achieved high precision (96%, 88%, 100%, and 97%, respectively). Most importantly, as I also discuss in §subsubsection 4.4.3, the technique is designed to (1) initially achieve high recall, possibly at the cost of low precision, and (2) improve precision by aggressively filtering the initial set of rules.

I discuss two examples of false positives and false negatives in the rules computed by NLP-based Rule Extraction. The first example is a false positive for the OhSome service, whose `GET/elements/area` operation has a parameter called `filter`. By analyzing the parameter description “Combines several attributive filters, e.g. OSM type, the geometry (simple feature) type, as well as the OSM tag; no default value”, NLP-based Rule Extraction identifies “OSM” as an example value for the parameter, creating the rule “`example: OSM`”. This is incorrect because “OSM” is an acronym for OpenStreetMap; that is, the parameter description is not providing literal examples for the OSM attributes that could be provided as filters, so no actual example can be extracted from the description. To address such cases, NLP-based Rule Extraction would have to leverage additional sources of domain-specific information. For instance, in this case, the OpenAPI specification of `graphhopper.com` contains relevant example values for OSM types (e.g., “node”, “way”, and “relation”).

The second example is a false negative for the FDIC service. The `GET/institutions` operation has a parameter `sort_order` with description “Indicator if ascending (ASC) or descending (DESC)”. NLP-based Rule Extraction failed to identify “ASC” or “DESC” as example values. The reason is that the search terms used for vocabulary terms identification were inadequate for detecting this sentence as potentially containing a parameter-example rule. This limitation could be addressed in different ways, such as expanding the set of search terms used for vocabulary terms identification and/or fine-tuning the `restW2V` model

with a larger training dataset.

NLP-based Rule Extraction effectively infers rules from OpenAPI specifications with high recall (88%–100%, 94% on average). However, its precision is lower and varies more across services (16%–100%, 50% on average), supporting the need of a validation phase to reduce false positives.

RQ2: Effectiveness of Rule Validation

Columns 8–14 of Table 4.2 present data on the effectiveness of Rule Validation in terms of the six metrics computed on the validated rules. For precision, recall, and F_1 score the table also shows, in parentheses, the percentage difference from the corresponding scores before validation.

Overall, the data show that Rule Validation is effective in improving the accuracy of NLP2REST. Across services, Rule Validation filtered out 235 of the 314 false positives, causing the precision to increase from 50% to 79%—a 58% improvement. Moreover, this was accompanied by only a small reduction in recall from 94% to 91% (i.e., 9 additional false negatives). It is worth noting that the decrease in recall occurs mostly due to inconsistencies between OpenAPI specifications and API implementations. In general, the validation phase results in an enhanced specification that is more aligned with the service implementation and is, therefore, more helpful to test generators in crafting successful requests.

It is worth mentioning that, although rule validation is effective in improving the precision of rule extraction, the effective of its dynamic part is limited when Rule Validation fails to produce successful requests. Consider, for instance, the incorrect rule `example: OSM` inferred by NLP-based Rule Extraction for the OhSome service. Rule Validation was unable to remove this incorrect rule because it failed to generate a valid request for the `GET /elements/area` operation, which contained the parameter.

Table 4.3: Improvement in performance of the testing tools considered when fed with enhanced specifications, compared to their baseline performance with original specifications.

Tool Name	Freq. of 2XX		Freq. of 4XX		Freq. of 5XX	
	Original	Enhanced	Original	Enhanced	Original	Enhanced
bBOXRT	11.5%	20.6% (+79%)	87.4%	77.9% (-11%)	1.1%	1.6% (+36%)
EvoMaster	14.3%	21.5% (+50%)	67.2%	54.5% (-23%)	18.5%	24.0% (+30%)
Morest	5.5%	8.3% (+51%)	73.4%	72.8% (-1%)	21.1%	18.9% (-10%)
REStest	55.2%	61.6% (+12%)	34.2%	29.6% (-13%)	10.6%	8.9% (-17%)
REStler	14.6%	11.3% (-23%)	50.4%	47.3% (-6%)	35.0%	41.4% (+18%)
RestTestGen	29.2%	32.1% (+10%)	68.4%	67.8% (-1%)	2.5%	0.1% (-96%)
Schemathesis	31.2%	36.9% (+18%)	52.5%	49.8% (-5%)	16.3%	13.3 (-18%)
Tcases	12.5%	17.7% (+42%)	46.0%	47.9% (-4%)	41.5%	40.8% (-2%)
Average	21.8%	26.2% (+20%)	59.9%	56.0% (-7%)	18.3%	18.6% (+2%)

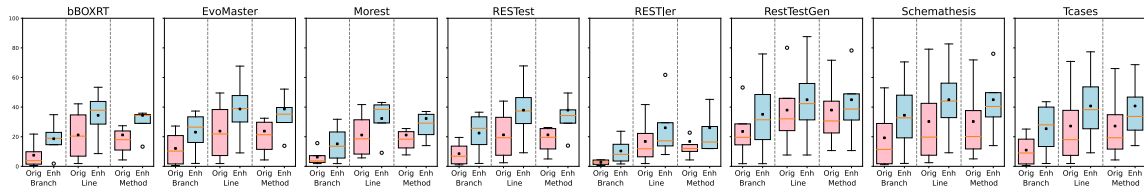


Figure 4.4: Comparison of branch, line, and method coverage results for original (Org) and enhanced (Enh) specifications.

Rule Validation eliminates $\sim 75\%$ of the false positives (235 out of 314), increasing precision by 58% (from 50% to 79%), while only causing a 3% decrease in recall.

RQ3: Comparison with State-of-the-art NLP-based Rule Extractors

I compared NLP2REST with RestCT, a REST API test generation tool that uses pattern matching to extract inter-parameter dependencies from text descriptions. Note that, due to compatibility issues with most of the evaluated REST APIs, RestCT was excluded from the RQ4-related experiment. However, these issues did not involve RestCT’s NLP module for inter-parameter dependencies detection, so I was able to use it to investigate RQ3.

I compared NLP2REST with RestCT’s NLP module by running both of them on the nine services in the benchmark and counting the number of inter-parameter dependency rules each tool detected. Because RestCT is based on pattern-matching alone, it was unable to identify any inter-parameter dependencies in the benchmark. In contrast, NLP2REST

uses constituency parse tree analysis to perform a more flexible matching, which allowed it to identify 15 out of the 19 inter-parameter dependencies in the benchmark.

NLP2REST is more effective than RestCT, a state-of-the-art NLP-based approach for OpenAPI rule extraction; NLP2REST was able to identify 15 out of 19 inter-parameter dependencies, whereas RestCT was unable to identify any.

RQ4: Improving Testing Tools Performance

Table 4.3 and Figure 4.4 present data on how the REST API testing tools perform when provided with the original and the enhanced OpenAPI specifications: Table 4.3 shows the frequency of status codes obtained, whereas Figure 4.4 shows the line, branch, and method coverage achieved. (As mentioned earlier, I performed coverage measurement only for the four open-source services in the benchmark.)

Table 4.3 shows, for each tool and the original and enhanced specifications, the ratio of the number of successful requests to the total number of requests sent (Freq. of 2XX), the ratio of the number of rejected requests to the total number of requests sent (Freq. of 4XX), and the ratio of requests triggering server errors to the total number of requests sent (Freq. of 5XX). The results show that, on average, enhanced specifications helped increase the rate of successful requests considerably (+20%) and server error responses slightly (+2%). They also helped decrease the rate of rejected requests moderately (-7%). For 5XX responses, I also computed the number of unique responses returned by the API. On average, the number of unique server errors increased by 4%, from 56.8 to 59. I note that the limited increase in the number of errors generated is partly due to the way in which the dynamic checking phase treats server error responses (5XX), which I plan to revisit in future work, as I discussed in §subsubsection 4.3.4.

Figure 4.4 depicts the comparison of branch, line, and method coverage achieved by each testing tool using the original (Orig) and enhanced (Enh) specifications. The figure is

divided into subplots, one for each tool, where each subplot compares the original and enhanced specifications for the three coverage types. The data is presented in the form of box plots that show the four quartiles along with the minimum, maximum, and outlier values. The mean value is shown as a (black) square marker, whereas the median value is depicted with a (orange) line. The results illustrate that the NLP2REST-enhanced specifications led to a significant improvement in coverage across all the tools considered, with average improvements of 103% (from 11.35% to 23.10%) for branch coverage, 50% (from 24.96% to 37.52%) for line coverage, and 52% (from 22.13% to 33.54%) for method coverage.

Overall, these results demonstrate the usefulness of the enhanced specifications generated by NLP2REST in improving the performance of REST API testing tools. The increase in both the number of successful requests and code coverage suggest that the rules generated by NLP2REST can have a significant impact on the performance of test generation tools.

The enhanced specifications generated by NLP2REST significantly improve the performance of REST API testing tools, helping them (1) generate a larger number of successful requests (+20%) and fewer bad requests (-7%), (2) trigger more unique server responses (+4%), and (3) achieve higher branch (+103%), method (+52%), and line (+50%) coverage.

4.4.4 Threats to Validity

Like all empirical evaluations, this study may suffer from external and internal threats to validity.

External Validity

The main threat to external validity for this evaluation is that the services and testing tools I considered might not be representative, and the results may therefore not generalize. To

mitigate this threat, I selected nine services based on two existing comprehensive studies on REST API testing, one focusing on commercial REST APIs [140] and the other on open-source REST APIs [141]. Furthermore, I selected eight testing tools based on the results presented in [141] and more recent publications. Additionally, the settings used for the testing tools and RESTful services might also not be optimal. However, I attempted to minimize the impact of hardware and software configurations on the results by using the same settings as a previous study [141] and default settings for all RESTful services during the experiments.

Internal Validity

A first internal threat to the validity of this evaluation is that the training set for the restW2V model might not be representative, which could impact the accuracy of the model. To mitigate this threat, I tried to make the training dataset as large and diverse as possible by using 1,875,607 text sets taken from 4,064 REST API specifications.

Another possible internal threat is the manual creation of the ground truth, as the individual interpretation of the specifications by the students involved could have affected the results. To address this, the students were asked to reach a consensus on any discrepancies, and they were not made aware of the specific rules elicited in this preliminary experiment, thus reducing the risk of bias.

Lastly, the experiment results reported in this chapter might also be influenced by errors in the implementation of the NLP2REST technique. To address this threat, I thoroughly examined and tested the code and made it publicly accessible to allow for review (as well as extensions) by others.

4.5 Conclusion

I presented NLP2REST, a novel approach that (1) extracts information from the human-readable parts of OpenAPI specifications, (2) uses the extracted information to enhance the

specifications, and (3) feeds the enhanced specification to (specification-based) REST API testing tools to improve their performance. The empirical evaluation show that NLP2REST is effective and accurate in generating enhanced specifications and that these enhanced specifications can indeed benefit testing tools by allowing them to generate larger numbers of valid requests, trigger more unique server responses, and achieve higher coverage.

CHAPTER 5

LEVERAGING LARGE LANGUAGE MODELS TO IMPROVE REST API TESTING

5.1 Introduction

This chapter introduces RestGPT, a new approach that harnesses Large Language Models (LLMs) to enhance REST API specifications by identifying constraints and generating relevant parameter values. Given an OpenAPI Specification [6], the most popular REST API specification language, ¹ RestGPT augments it by deriving constraints and example values. Existing approaches such as NLP2REST [144] require a validation process to improve precision, which involves not just the extraction of constraints but also executing requests against the APIs to dynamically check these constraints. Such a process demands significant engineering effort and a deployed service instance, making it cumbersome and time-consuming. In contrast, RestGPT achieves higher precision without requiring expensive validation. Furthermore, unlike ARTE [50], RestGPT excels in understanding the context of a parameter name based on an analysis of the parameter description, thus generating more contextually relevant values.

The preliminary results demonstrate the significant advantage of this approach over existing tools. Compared to NLP2REST without the validation module, this method improves precision from 50% to 97%. Even when compared to NLP2REST equipped with its validation module, this approach still increases precision from 79% to 97%. Additionally, RestGPT successfully generates both syntactically and semantically valid inputs for 73% of the parameters over the analyzed services and their operations, a considerable improvement over ARTE, which could generate valid inputs for 17% of the parameters only.

¹OpenAPI specification, previously known as Swagger [7], is the most popular REST API specification language; other specification languages include RAML [21] and API Blueprint [22].

```

/institutions:
  get:
    operationId: searchInstitutions
    produces:
      - application/json
    parameters:
      - name: filters
        in: query
        required: false
        type: string
        description: The filter for the bank search.
          Examples:
            * Filter by State name
            `STNAME:"West Virginia"`
            * Filter for any one of multiple State names
            `STNAME:(\"West Virginia\", \"Delaware\")`
      - name: sort_order
        in: query
        required: false
        type: string
        description: Indicator if ascending (ASC) or descending (DESC)
    responses:
      '200':
        description: successful operation
        schema:
          type: object

```

Figure 5.1: A part of FDIC Bank Data’s OpenAPI specification.

Given these encouraging results, I outline a number of research directions for leveraging LLMs in other ways for further enhancing REST API testing. These findings demonstrate the effectiveness of RestGPT and highlight the potential of leveraging LLMs for improving REST API testing. To facilitate reproducibility and further exploration, all the code, data, and results are made available in the artifact repository [[RestGPTartifact](#)].

5.2 Motivating Example

The OpenAPI specification for the Federal Deposit Insurance Corporation (FDIC) Bank Data’s API, shown in Figure 5.1, serves to offer insights into banking data. Using this example, I highlight the challenges in parameter value generation faced by current REST API testing assistant tools and illustrate how RestGPT addresses these challenges.

1. **Parameter filters:** Although the description provides guidance on how the parameter should be used, ARTE’s dependency on DBpedia results in no relevant value generation for `filters`. NLP2REST, with its keyword-driven extraction, identi-

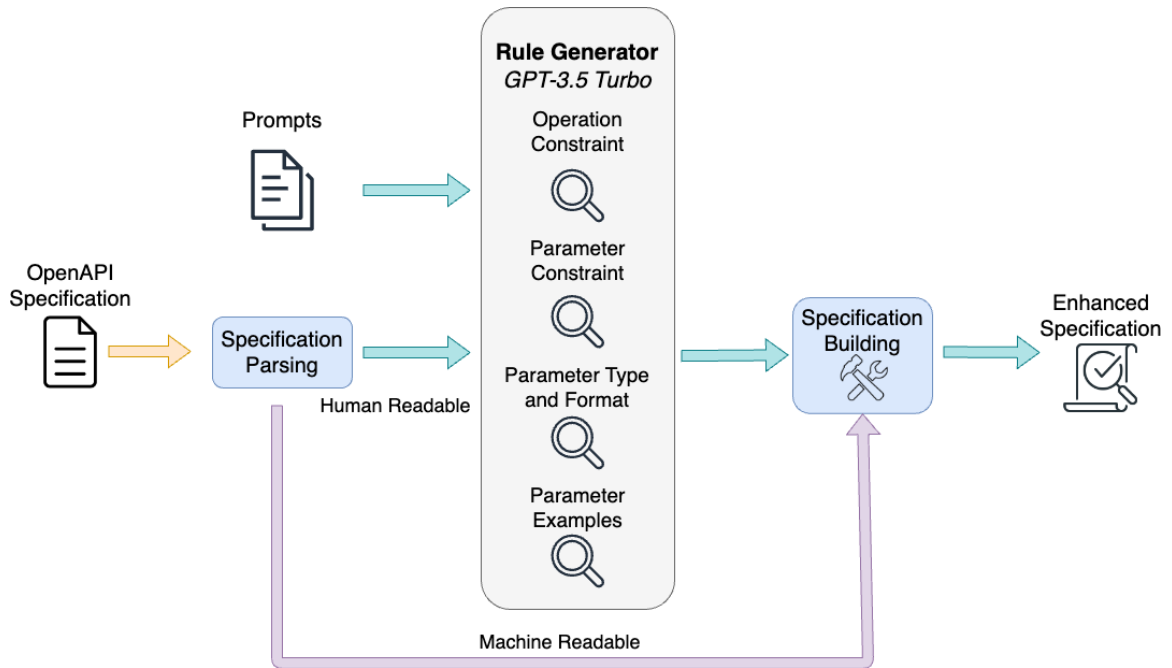


Figure 5.2: Approach Overview.

fies examples from the description, notably aided by the term “example”. Consequently, patterns such as `STNAME: "West Virginia"` and `STNAME: ("West Virginia", "Delaware")` are accurately captured.

2. **Parameter sort_order:** Here, both tools exhibit limitations. ARTE, while querying DBPedia, fetches unrelated values such as “List of colonial heads of Portuguese Timor”, highlighting its contextual inadequacy. In the absence of identifiable keywords, NLP2REST fails to identify “ASC” or “DESC” as potential values.

In contrast to these tools, RestGPT is much more effective: with a deeper semantic understanding, RestGPT accurately discerned that the `filters` parameter was contextualized around state names tied to bank records, and generated test values such as `STNAME: "California"` and multi-state filters such as `STNAME: ("California", "New York")`. Also, it successfully identifies the values “ASC” or “DESC” from the description of the `sort_order` parameter. This example illustrates RestGPT’s superior contextual understanding, which enable it to outperform the constrained or context-blind

Table 5.1: Effectiveness of NLP2REST and RestGPT.

REST Service	No. of Rules in Ground Truth	NLP2REST Without Validation Process						NLP2REST With Validation Process						RestGPT					
		TP	FP	FN	Precision	Recall	F ₁	TP	FP	FN	Precision	Recall	F ₁	TP	FP	FN	Precision	Recall	F ₁
FDIC	45	42	36	3	54%	93%	68%	42	25	3	63%	93%	75%	44	0	1	100%	98%	99%
Genome Nexus	81	79	3	2	96%	98%	97%	79	3	2	96%	98%	97%	75	0	6	100%	93%	96%
LanguageTool	20	20	12	0	63%	100%	77%	18	2	2	90%	90%	90%	18	0	3	100%	86%	92%
OCVN	17	15	2	2	88%	88%	88%	13	1	4	93%	76%	84%	15	2	1	88%	94%	91%
OhSome	14	13	66	1	16%	93%	28%	12	11	2	52%	80%	63%	12	3	2	80%	86%	83%
OMDb	2	2	0	0	100%	100%	100%	2	0	0	100%	100%	100%	2	0	0	100%	100%	100%
REST Countries	32	28	1	4	97%	88%	92%	28	0	4	100%	88%	93%	30	0	2	100%	94%	97%
Spotify	88	83	68	5	55%	94%	69%	82	28	6	75%	93%	83%	86	2	4	98%	96%	97%
YouTube	34	30	126	4	19%	88%	32%	28	9	6	76%	82%	79%	24	2	8	92%	75%	83%
Total	333	312	314	21	50%	94%	65%	304	79	29	79%	91%	85%	306	9	27	97%	92%	94%

methodologies of existing tools.

5.3 Approach

This section details the methodology for harnessing LLMs to enhance OpenAPI specifications, encompassing inputs, processes, and expected outputs as depicted in Figure 5.2.

Table 5.2: Accuracy of ARTE and RestGPT.

Service Name	ARTE	RestGPT
FDIC	25.35%	77.46%
Genome Nexus	9.21%	38.16%
Language-Tool	0%	82.98%
OCVN	33.73%	39.76%
OhSome	4.88%	87.80%
OMDb	36.00%	96.00%
REST-Countries	29.66%	92.41%
Spotify	14.79%	76.06%
Youtube	0%	65.33%
Average	16.93%	72.68%

5.3.1 Overview

As depicted in Figure 5.2, the RestGPT workflow begins with parsing input OpenAPI specifications. This step identifies each parameter’s machine-readable and human-readable sections. The human-readable section yields four critical types of constraints: operational constraints, parameter constraints, parameter type and format, and parameter examples. These classifications stem from the recent work [144] that can significantly enhance testing tools’ ability. Rule Generator generate those four rules with a set of prompts. It uses GPT-3.5 Turbo with its remarkable speed and performance as shown in a recent report[24].

The incorporation of few-shot learning amplifies this benefit. By presenting the LLM with succinct, contextually rich instructions and examples, few-shot prompts ensure that outputs resonate with relevance and precision [25, 145]. Finally, using the generated rules and the original Specification, REST GPT generates enhanced specification.

It is worth noting that unrelated outputs are mitigated through the incorporation of a low temperature value of 0.2, purposeful prompt-design that addresses vague inputs, and a refinement step where unexpected symbols and outputs are removed. Finally, these refined rules are amalgamated with the machine-readable segment of the specification, ensuring the resulting in a more comprehensive and enhanced specification.

Accurate rule extraction plays a pivotal role in enhancing API specifications. Crafting precise prompts for few-shot learning becomes a linchpin in achieving this accuracy, given the sensitivity of performance to prompt design [146]. Drawing upon best practices from reputable learning materials [147, 148], the few-shot prompts are designed to ensure clarity and accuracy in rule interpretation.

The Rule Generator is the core module of RestGPT, responsible for crafting prompts that instruct the model on rule interpretation, and specifying the output formatting for future parsing. Given the key role of precise prompts in few-shot learning [146], I have drawn upon best practices on prompt crafting (e.g., [147, 148]). These guidelines serve as primary instructions for the model, ensuring that the extracted rules are both accurate and relevant. The cases provide specific scenarios and expected outputs to guide the model's behavior during rule extraction. The Grammar Highlights emphasize key operators that the model should recognize and employ during rule generation.

5.3.2 Rule Generator

The Rule Generator is the core module of RestGPT, tasked with prompts that instruct the model on rule interpretation and establishing guidelines to govern the form and content of the outputs. The module guides the model to accurately identify and categorize rules,

ensuring that the responses are not only accurate but also structured for seamless integration in subsequent processes. Given the key role of precise prompts in few-shot learning [146], I have drawn upon best practices on prompt crafting (e.g., [147, 148]).

To best instruct the model on rule interpretation and output formatting, the prompts are designed around four core components: guidelines, cases, grammar highlights, and output configurations.

Guidelines

1. Identify the parameter using its name and description.
2. Extract logical constraints from the parameter description, adhering strictly to the provided format.
3. Interpret the description in the least constraining way.

The provided guidelines serve as the foundational instructions for the model, framing its perspective and clarifying its primary objectives. Using the guidelines as a basis, RestGPT can then proceed with more specific prompting.

Cases

- Case 1:** If the description is non-definitive about parameter requirements: Output "None".
- ...
- Case 10:** For complex relationships between parameters: Combine rules from the grammar.

The implementation of cases in model prompting plays a pivotal role in directing the model's behaviour, ensuring that it adheres to precise criteria as depicted in the example. Drawing inspiration from Chain-of-Thought prompting [39], I decompose rule extraction into specific, manageable pieces to mitigate ambiguity and, consequently, improve the model's processing abilities.

OpenAPI Grammar Highlights

Relational Operators: '<', '>', '<=', '>=', '==', '! ='

Arithmetic Operators: '+', '-', '*', '/'

Dependency Operators: 'AllOrNone', 'ZeroOrOne', ...

The OpenAPI Grammar Highlights emphasize key operators and vocabulary that the model should recognize and employ during rule extraction. By providing the model with a fundamental context-specific language, RestGPT identifies rules within text.

Output Configurations

Example Parameter Constraint: min [minimum], max [maximum], default [default]

Example Parameter Format: type [type], items [item type], format [format], collectionFormat [collectionFormat]

After guiding the model through the rule-extraction process via specific prompting, I lastly define output formatting to compile the model's findings into a simple structure for subsequent processing. Additionally, the Rule Generator also oversees the value-generation process, which is executed during the extraction of parameter example rules. The artifact [**RestGPTartifact**] provides details of all the prompts and their corresponding results.

5.3.3 Specification Enhancement

The primary objective of RestGPT is to improve the effectiveness of REST API testing tools. I accomplish this by producing enhanced OpenAPI specifications, augmented with rules derived from the human-readable natural-language descriptions in conjunction with the machine-readable OpenAPI keywords [149].

As illustrated in Figure 5.2, the *Specification Parsing* stage extracts the machine-readable and human-readable components from the API specification. After rules from

the natural language inputs have been identified by the *Rule Generator*, the *Specification Building* phase begins. During this phase, the outputs from the model are processed and combined with the machine-readable components, ensuring that there is no conflict between restrictions. For example, the resulting specification must have the `style` attribute only if the data type is `array` or `object`. The final result is an enriched API specification that contains constraints, examples, and rules extracted from the human-readable descriptions, enabling the subsequent REST API testing tool to perform more robust and informed testing.

5.4 Preliminary Results

To measure the effectiveness of RestGPT in REST API rule generation, I conduct a comparative analysis against two renowned works: NLP2REST [144] and ARTE [50]. This section introduces the evaluation methodology and insights gained from this evaluation.

5.4.1 Evaluation Methodology

I collected nine RESTful services from the NLP2REST study. The motivation behind this selection is the availability of a ground truth of extracted rules in the NLP2REST work [144]. Having this data, I could easily compare this work with NLP2REST.

To establish a comprehensive benchmark, I incorporated a comparison with ARTE as well. This approach was guided by the ARTE paper, from which I extracted the necessary metrics for comparison. Adhering to ARTE’s categorization of input values as Syntactically Valid and Semantically Valid [50], two PhD students meticulously verified the input values generated by RestGPT and ARTE. Notably, I emulated ARTE’s approach in scenarios where more than ten values were generated by randomly selecting ten from the pool for analysis.

5.4.2 Results and Discussion

Table 5.1 presents a comparison of the rule-extraction capabilities of NLP2REST and RestGPT. RestGPT excels in precision, recall, and the F_1 score across a majority of the REST services. NLP2REST, while effective, hinges on a validation process that involves evaluating server responses to filter out unsuccessful rules. This methodology demands engineering effort, and its efficacy is constrained by the validator’s performance.

In contrast, RestGPT eliminates the need for such validation entirely with its high precision. Impressively, RestGPT’s precision of 97% surpasses even the precision of NLP2REST post-validation, which stands at 79%. This emphasizes that RestGPT is able to deliver superior results without a validation stage. This result shows an LLM’s superior ability in nuanced rule detection, unlike conventional NLP techniques that rely heavily on specific keywords.

Furthermore, Table 5.2 presents data on accuracy of ARTE and RestGPT. The data paint a clear picture: RestGPT consistently achieves higher accuracy than ARTE across all services. This can be attributed to the context-awareness capabilities of LLMs, as discussed in Section section 5.2. For example, in language-tool service, I found that, for the `language` parameter, ARTE generates values such as “Arabic”, “Chinese”, “English”, and “Spanish”. However, RestGPT understands the context of the language parameter, and generates language code such as “en-US” and “de-DE”. I believe this superior performance will aid REST API testing tools more effectively. Additionally, it underscores the promising potential of LLMs in this domain.

5.4.3 Threats to Validity

There exist a few threats that could impact the validity of this approach and findings. First, the closed-source nature of the LLM used introduces validity threats, since I have no control over or complete transparency into model evolution which could render findings obsolete.

Additionally, on the same input, the model output can vary and thus replicating results

exactly is not always feasible. Future experiments could benefit from output variability studies on the parameters and rules generated.

Second, I evaluated RestGPT using one particular LLM. Testing other LLMs is required to confirm overall approach viability rather than with only one model. Future work should repeat experiments using various open-source models to enable comparison and reduce the impact of closed-source model evolution resulting in differing outputs. Also, the hallucination problem inherent to LLMs presents a threat to the validity of this approach. LLMs are known to sometimes generate plausible but incorrect or irrelevant information, which can affect the accuracy of the generated OpenAPI rules.

Moreover, the preliminary evaluation was conducted on only nine REST services. More publicly available and industrial APIs should be analyzed to establish broader validity. The findings may also not generalize beyond the services studied here. Testing on larger scale commercial systems is also warranted.

Finally, the result evaluation for comparison with NLP2REST and ARTE was conducted with manual counting techniques. Future approaches could benefit from the development of automated and standardized benchmarks gauging rule and parameter accuracy.

5.5 Conclusion

I introduced RestGPT, an innovative tool that leverages the capabilities of LLMs to improve REST API testing. While current automated techniques fail to achieve high levels of code coverage due to their inability to grasp the nuances of REST API specifications, RestGPT adeptly combines the strengths of ARTE and NLP2REST. This preliminary evaluation provides initial, yet clear evidence of its superiority over existing techniques in terms of precision and both syntactic and semantic accuracy. Future plans for RestGPT include model fine-tuning and optimization. Moreover, I will implement LLM-powered bug detection and introduce an LLM-based testing approach that incorporates server response analysis using LLMs. By pursuing this line of research, I believe that I can make substan-

tial contributions to the area of REST API testing. It not only interprets the semantics from parameter names but also recognizes constraints through profound semantic understanding. Preliminary evaluations underscore its superiority over existing tools in terms of precision and both syntactic and semantic accuracy.

CHAPTER 6

ADAPTIVE REST API TESTING WITH REINFORCEMENT LEARNING

6.1 Introduction

Testing REST APIs can be challenging because of the large search space to be explored, due to the large number of operations, potential execution orders, inter-parameter dependencies, and associated input parameter value constraints [19, 106]. Current techniques often struggle when exploring this space due to lack of effective exploration strategies for operations and their parameters. In fact, existing testing tools tend to treat all operations and parameters equally, disregarding their relative importance or complexity, which leads to suboptimal testing strategies and insufficient coverage of crucial operation and parameter combinations. Moreover, these tools rely on discovering producer-consumer relationships between response schemas and request parameters; this approach works well when the parameter and response schemas are described in detail in the specification, but falls short in the common case in which the schemas are incomplete or imprecise.

To address these limitations of the state of the art, I present adaptive REST API testing with reinforcement learning (ARAT-RL), an advanced black-box testing approach. This technique incorporates several innovative features, such as leveraging reinforcement learning to prioritize operations and parameters for exploration, dynamically constructing key-value pairs from both response and request data, analyzing these pairs to inform dependent operations and parameters, and utilizing a sampling-based strategy for efficient processing of dynamic API feedback. The primary objectives of this approach are to increase code coverage and improve fault-detection capability.

REST API testing presents a unique challenge that aligns well with reinforcement learning paradigms. The process of exploring an API can be naturally modeled as a sequential

decision-making problem where each action (API call) affects the environment (API state) and provides feedback (response). This sequential nature, combined with delayed rewards (coverage improvements may only be observable after multiple related API calls), makes reinforcement learning particularly suitable for this domain.

I chose Q-learning specifically for ARAT-RL because it offers several advantages for REST API testing. First, Q-learning is a model-free approach that doesn't require prior knowledge of environment dynamics, making it well-suited for black-box API testing where the internal behavior is unknown. Second, its off-policy nature allows for exploration of the API space while simultaneously learning an optimal testing strategy. Third, Q-learning's value-based approach enables straightforward prioritization of operations and parameters based on their estimated utility, which directly addresses the need to differentiate between more and less important API components. Finally, Q-learning's simplicity and efficiency make it practical for real-time adaptation during the testing process, with minimal computational overhead compared to more complex reinforcement learning algorithms. This balance of exploration capability, adaptability, and computational efficiency makes Q-learning particularly appropriate for addressing the challenges of REST API testing.

The core novelty in ARAT-RL is an adaptive testing strategy driven by a reinforcement-learning-based prioritization algorithm for exploring the space of operations and parameters. The algorithm initially determines the importance of an operation based on the parameters it uses and the frequency of their use across operations. This targeted exploration enables efficient coverage of critical operations and parameters, thereby improving code coverage. The technique employs reinforcement learning to adjust the priority weights associated with operations and parameters based on feedback, by decreasing importance for successful responses and increasing it for failed responses. The technique also assigns weights to parameter-value mappings based on various sources of input values (e.g., random, specified values, response values, request values, and default values). The adaptive

nature of ARAT-RL gives precedence to yet-to-be-explored or previously error-prone operations, paired with suitable value mappings, which improves the overall efficiency of API exploration.

Another innovative feature of ARAT-RL is dynamic construction of key-value pairs. In contrast to existing approaches that rely heavily on resource schemas provided in the specification, this technique dynamically constructs key-value pairs by analyzing POST operations (i.e., the HTTP methods that create resources) and examining both response and request data. For instance, suppose that an operation takes book title and price as request parameters and, as response, produces a success status code along with a string message (e.g., “Successfully created”). This technique leverages this information to create key-value pairs for book title and price, upon receiving a successful response, even if such data is not present in the response. In other words, the technique takes into account the input parameters used in the request, as they correspond to the created resource. Moreover, if the service returns incomplete resources (i.e., only a subset of the data items for a given type of resource), this technique still creates key-value pairs for the information available. This dynamic approach enables ARAT-RL to identify resources from the API responses and requests, as well as discover hidden dependencies that are not evident from the specification alone.

Finally, ARAT-RL employs a simple yet effective sampling-based approach that allows it to process dynamic API feedback efficiently and adapt its exploration based on the gathered information. By randomly sampling key-value pairs from responses, this technique reduces the overhead of processing every response for each pair, resulting in more efficient testing.

To evaluate ARAT-RL, I conducted a set of empirical studies using 10 RESTful services and compared its performance against three state-of-the-art REST API testing tools: RESTler [9], EvoMaster [45], and Morest [17]. I assessed the effectiveness of ARAT-RL in terms of coverage achieved and service failures triggered, and its efficiency in terms of

valid and fault-inducing requests generated and operations covered within a given time budget. The results show that ARAT-RL outperforms the competing tools for all the metrics considered—it achieved the highest method, branch, and line coverage rates, along with better fault-detection ability. Specifically, ARAT-RL covered 119%, 60%, and 52% more branches, lines, and methods than RESTler; 37%, 21%, and 14% more branches, lines, and methods than EvoMaster; and 24%, 12%, and 10% more branches, lines, and methods than Morest. ARAT-RL also uncovered 9.3x, 2.5x, and 2.4x more bugs than RESTler, EvoMaster, and Morest, respectively. In terms of efficiency, ARAT-RL generated 52%, 41%, and 1,222% more valid and fault-inducing requests and covered 15%, 24%, and 283% more operations than Morest, EvoMaster, and RESTler, respectively. I also conducted an ablation study to assess the individual effects of prioritization, dynamic feedback analysis, and sampling on the overall effectiveness of ARAT-RL. The results indicate that reinforcement-learning-based prioritization contributes the most to ARAT-RL’s effectiveness, followed by dynamic feedback analysis and sampling.

The main contributions of the work in this chapter are:

- A novel approach for adaptive REST API testing that incorporates (1) reinforcement learning to prioritize exploration of operations and parameters, (2) dynamic analysis of request and response data to identify dependent parameters, and (3) a sampling strategy for efficient processing of dynamic API feedback.
- Empirical results demonstrating that ARAT-RL outperforms state-of-the-art REST API testing tools by generating more valid and fault-inducing requests, covering more operations, achieving higher code coverage, and triggering more service failures.
- An artifact [150] containing the tool, the benchmark services, and the empirical results.

The rest of this chapter is organized as follows. Section section 6.2 presents a moti-

```

/products/{productName}/configurations/{configurationName}/features/{featureName}:
  post:
    operationId: addFeatureToConfiguration
    produces:
      - application/json
    parameters:
      - name: productName
        in: path
        required: true
        type: string
      - name: configurationName
        in: path
        required: true
        type: string
      - name: featureName
        in: path
        required: true
        type: string
    responses:
      default:
        description: successful operation
/products/{productName}/configurations/{configurationName}/features:
  get:
    operationId: getConfigurationActivatedFeatures
    produces:
      - application/json
    parameters:
      - name: productName
        in: path
        required: true
        type: string
      - name: configurationName
        in: path
        required: true
        type: string
    responses:
      '200':
        description: successful operation
        schema:
          type: array
          items:
            type: string

```

Figure 6.1: A Part of Features Service’s OpenAPI Specification.

vating example to illustrate challenges in REST API testing. Section section 6.3 describes this approach. Section section 6.4 presents the empirical evaluation and results. Finally, Section section 6.5 presents the conclusions.

6.2 Motivating Example

Next, I illustrate the salient features of ARAT-RL using the Feature-Service specification (Figure 6.1) as an example.

RL-based adaptive exploration: For the example in Figure 6.1, to perform the operation

`addFeatureToConfiguration`, I must first create a product using a separate operation and establish a configuration for it using another operation. The sequence of operations should, therefore, be: create product, create configuration, and create feature name for the product with the specified configuration name. This example emphasizes the importance of determining the operation sequence. This technique initially assigns weights to operations and parameters based on their usage frequency in the specification. In this case, `productName` is the most frequently used parameter across all operations; therefore, this technique assigns higher weights to operations involving `productName`. Specifically, the operation for creating a product gets the highest priority.

Moreover, once an operation is executed, its priority must be adjusted so that it is not explored repeatedly, creating new product instances unnecessarily. After processing a prioritized operation, this technique employs RL to adjust the weights in response to the API response received. If a successful response is obtained, negative rewards are assigned to the processed parameters, as the objective is to explore other uncovered operations. This method naturally leads to the selection of the next priority operation and parameter, facilitating efficient adjustments to the call sequence.

Inter-parameter dependencies [106] can increase the complexity of the testing process, as some parameters might have mutual exclusivity or other constraints associated with them (e.g., only one of the parameters can be specified). RL-based exploration guided by feedback received can also help with dealing with this complexity.

Dynamic construction of key-value pairs: Existing REST API testing strategies (e.g., [8, 9, 17]) emphasize the importance of identifying producer-consumer relationships between response schemas and request parameters. However, current tools face limitations when operations produce unstructured output (e.g., plain text) or have incomplete response schemas in their specifications. For instance, the `addFeatureToConfiguration` operation lacks structured response data (e.g., JSON format). Despite this, this approach processes and generates key-value data `{productName: <value>, configurationName: <value>,`

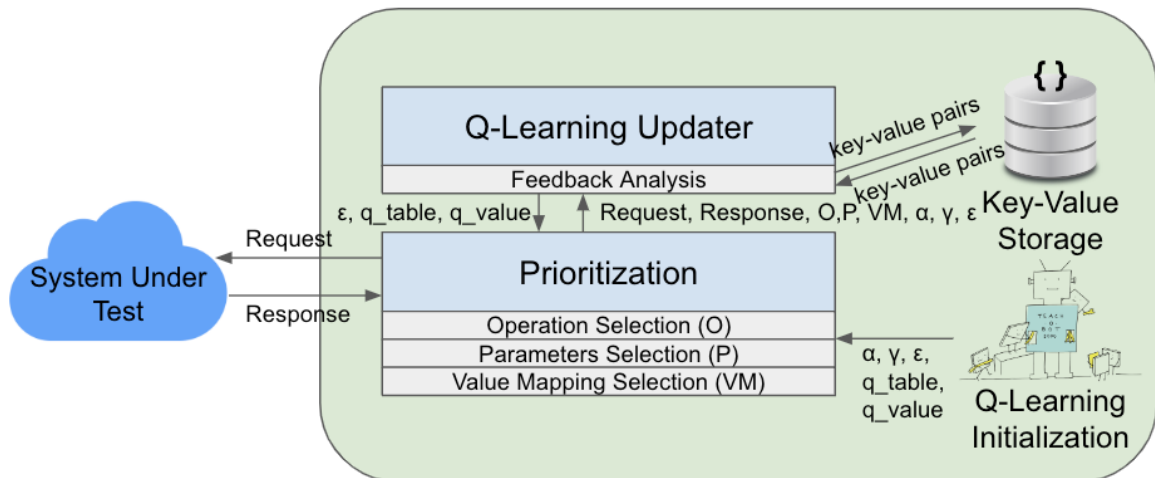


Figure 6.2: Approach Overview

featureName: <value>} from the request data, as the POST HTTP method indicates that a resource is created using the provided inputs.

By analyzing and storing key-value pairs identified from request and response data, this dynamic key-value pair construction method proves especially beneficial in cases of responses with plain-text descriptions or incomplete response schemas. The technique can effectively uncover hidden dependencies not evident from the specification.

Sampling for efficient dynamic key-value pair construction: API response data can sometimes be quite large and processing every response for each key-value pair can be computationally expensive. To address this issue, I have incorporated a sampling strategy into this dynamic key-value pair construction method. This strategy efficiently processes the dynamic API feedback and adapts its exploration based on the gathered information while minimizing the overhead of processing every response.

6.3 Approach

In this section, I introduce the Q-Learning-based REST API testing approach, which intelligently prioritizes and selects operations, parameters, and value-mapping sources while dynamically constructing key-value pairs. Figure 6.2 provides a high-level overview of this

approach. Initially, the Q-Learning Initialization module sets up the necessary variables and tables for the Q-learning process. Q-Learning Updater subsequently receives these variables and tables and passes them to the Prioritization module, which is responsible for selecting operations, parameters, and value-mapping sources.

Q-Learning is a type of reinforcement learning algorithm that is used to optimize decision-making over time by learning from interactions with an environment. It is a **model-free, value-based** algorithm that aims to learn the optimal action-selection policy by estimating the value of state-action pairs (called Q-values). The algorithm updates Q-values iteratively using the formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

Where:

- s is the current state,
- a is the action taken,
- r is the reward received after taking action a ,
- α is the learning rate, and
- γ is the discount factor that controls the importance of future rewards.

Traditional REST API testing tools typically rely on HTTP status codes (e.g., 2xx for success, 4xx for failure) to determine whether a request succeeded or failed, but they often waste a significant number of requests because they do not leverage the outcomes of previous requests to guide future ones. This lack of learning leads to inefficiencies, especially in large and complex API spaces.

Q-learning, being a value-driven algorithm, allows me to **learn from previous interactions** with the API. By associating positive rewards with successful requests (e.g., receiving a 2xx status code) and negative rewards with failed ones (e.g., receiving a 4xx status code),

the algorithm can prioritize API requests that are more likely to succeed based on past experiences. This adaptive behavior significantly reduces wasted requests and improves overall testing efficiency.

Additionally, Q-learning's ability to **explore and exploit** makes it ideal for REST API testing, where the environment is partially unknown, and there's a need to balance between trying new requests and reusing previously successful strategies. Unlike other reinforcement learning models, Q-learning is particularly suited for this task because it doesn't require a model of the environment and can directly work with the feedback (HTTP responses) from the System Under Test (SUT).

ARAT-RL then sends a request to the System Under Test (SUT) and receives a response. It also supplies the request, response, selected operation, parameters, mapped value source, and the Q-Learning parameters (α , γ , and ϵ) to Q-Learning Updater. The feedback is analyzed with the request and response, storing key-value pairs extracted from them for future use. The Updater component then adjusts the Q-values based on the outcomes, enabling the approach to adapt its decision-making process over time. ARAT-RL iterates through this procedure until the specified time limit is reached. In the rest of this section, I present the details of the algorithm.

6.3.1 Q-Learning Table Initialization

The Q-Learning Table Initialization component, shown in Algorithm Algorithm 4, is responsible for setting up the initial Q-table and Q-value data structures that guide the decision-making process throughout a testing session. Crucially, this process happens without making any API calls.

The algorithm begins by setting the learning rate α to 0.1, the discount factor γ to 0.99, and the exploration rate ϵ to 0.1 (lines 2–4). These parameters control the learning and exploration process of the Q-Learning algorithm; the chosen values are the ones that are commonly recommended and used (e.g., [151, 152, 153]). The algorithm then initializes

Algorithm 4 Q-Learning Table Initialization

```
1: procedure INITIALIZEQLEARNING(operations)
2:   Set learning rate ( $\alpha$ ) to 0.1
3:   Set discount factor ( $\gamma$ ) to 0.99
4:   Set exploration rate ( $\epsilon$ ) to 0.1
5:   Initialize empty dictionary q_table
6:   Initialize empty dictionary q_value
7:   for operation in operations do
8:     operation_id  $\leftarrow$  operation['operationId']
9:     q_value[operation_id]  $\leftarrow$  new dictionary
10:    for source in [S1, S2, S3, S4, S5] do
11:      q_value[operation_id][source]  $\leftarrow$  0
12:    end for
13:    for parameter in operation['parameters'] do
14:      param_name  $\leftarrow$  parameter['name']
15:      if key in q_table then
16:        q_table[param_name] = q_table[param_name] + 1
17:      else
18:        q_table[param_name] = 1
19:      end if
20:    end for
21:    for response_data in operation_data.get('responses') do
22:      for key in response_data.keys() do
23:        if key in q_table then
24:          q_table[key] = q_table[key] + 1
25:        end if
26:      end for
27:    end for
28:  end for
29:  return  $\alpha, \gamma, \epsilon, q\_table, q\_value$ 
30: end procedure
```

the Q-table and the Q-value with empty dictionaries (lines 5–6).

The algorithm iterates through each operation in the API (lines 7–24). For each operation, it extracts the operation’s unique identifier (`operation_id`) and creates a new entry in the Q-value dictionary for the operation (lines 8–9). Next, it initializes the Q-value for each value-mapping source (S1–S5) to zero (lines 10–12).

The algorithm proceeds to iterate through each parameter in the operation (lines 13–20). It extracts the parameter’s name (`param_name`) and, if `param_name` already exists in the Q-table, increments the corresponding entry by one; otherwise, it initializes the entry to one. This step builds the Q-table with counts of occurrences of each parameter.

Next, the algorithm iterates through the response data of each operation (lines 21–27). It extracts keys from the response and checks, for each key, whether it is present in the Q-table for that operation (line 18). If a key is present, the algorithm increments the corresponding entry in the Q-table by one (lines 23–25). This step populates the Q-table

with the frequency of occurrence of each response key.

Finally, the algorithm returns the learning rate α , the discount factor γ , the exploration rate ϵ , the Q-table, and the Q-value (line 29). This initial setup provides the Q-Learning algorithm with basic information about the API operations and their relationships, which is further refined during testing.

6.3.2 Q-Learning-based Prioritization

In this step, ARAT-RL prioritizes API operations and selects the best parameters and value-mapping sources based on their Q-values. I present Algorithm 5 (*SelectOperation*, *SelectParameters*, and *SelectValueMappingSource*) to describe the prioritization process.

The *SelectOperation* procedure is responsible for selecting the best API operation to exercise next. The algorithm initializes variables to store the maximum average Q-value and the best operation (lines 2–3). It then iterates through each operation, calculating the average Q-value for the operation’s parameters (lines 4–17). The operation with the highest average Q-value is selected as the best operation (line 15).

The *SelectParameters* procedure selects a subset of parameters for the chosen API operation. This selection is guided by the exploration rate ϵ . If a random value is greater than ϵ , the algorithm selects the top n parameters sorted by their Q-values in descending order (lines 4–8); otherwise, it randomly selects n parameters from the operation’s parameters (lines 9–12). Finally, the selected parameters are returned (line 14).

The *SelectValueMappingSource* procedure is responsible for selecting the value-mapping source for the chosen API operation. The technique leverages five sources of values.

- Source 1 (example values in specification): These values are provided in the API documentation as examples for a parameter. I consider three types of OpenAPI keywords that can specify example values: `enum`, `example`, and `description` [6]. The

Algorithm 5 Q-Learning-based Prioritization

```
1: procedure SELECTOPERATION(operations, q_table)
2:   Initialize max_avg_q_value  $\leftarrow -\infty$ 
3:   Initialize best_operation  $\leftarrow$  None
4:   for operation in operations do
5:     operation_id  $\leftarrow$  operation['operationId']
6:     Initialize sum_q_value  $\leftarrow$  0
7:     Initialize num_params  $\leftarrow$  len(operation['parameters'])
8:     for parameter in operation['parameters'] do
9:       param_name  $\leftarrow$  parameter['name']
10:      sum_q_value  $\leftarrow$  sum_q_value + q_table[param_name]
11:    end for
12:    avg_q_value  $\leftarrow$  sum_q_value / num_params
13:    if avg_q_value > max_avg_q_value then
14:      max_avg_q_value  $\leftarrow$  avg_q_value
15:      best_operation  $\leftarrow$  operation
16:    end if
17:  end for
18:  return best_operation
19: end procedure

1: procedure SELECTPARAMETERS(operation,  $\epsilon$ )
2:   Set n randomly ( $0 \leq n \leq$  length of operation['parameters'])
3:   Initialize empty list selected_parameters
4:   if random.random() >  $\epsilon$  then
5:     Sort operation['parameters'] by Q-values in descending order
6:     for i  $\leftarrow$  0 to n - 1 do
7:       Append operation['parameters'][i] to selected_parameters
8:     end for
9:   else
10:    for param in random.sample(operation['parameters'], n) do
11:      Append param to selected_parameters
12:    end for
13:  end if
14:  return selected_parameters
15: end procedure

1: procedure SELECTVALUEMAPPINGSOURCE(operation,  $\epsilon$ )
   Source1: Example values in specification
   Source2: Random value generated based on parameter's type and format
   Source3: Dynamically constructed key-value pairs from request
   Source4: Dynamically constructed key-value pairs from response
   Source5: Default values (string: string, number: 1.1, integer: 1, array: [], object: {})

2:   operation_id  $\leftarrow$  operation['operationId']
3:   sources  $\leftarrow$  [S1, S2, S3, S4, S5]
4:   if random.random() >  $\epsilon$  then
5:     max_q_value  $\leftarrow -\infty$ 
6:     max_q_index  $\leftarrow -1$ 
7:     for s in sources do
8:       if q_value[operation_id][s] > max_q_value then
9:         max_q_value  $\leftarrow$  q_value[operation_id][s]
10:        max_q_index  $\leftarrow$  s
11:      end if
12:    end for
13:    return max_q_index
14:  else
15:    return random.randint(1, 5)
16:  end if
17: end procedure
```

Algorithm 6 Q-Learning-based API Testing

```
1: procedure QLEARNINGUPDATER(response, q_table, q_value, selected_op, selected_params,  $\alpha$ ,  $\gamma$ )
2:   operation_id  $\leftarrow$  selected_op[operation_id']
3:   if response.status_code is 2xx (successful) then
4:     Extract key-value pairs from request and response
5:     reward  $\leftarrow$  -1
6:     Update q_value negatively
7:   else if response.status_code is 4xx or 500 (unsuccessful) then
8:     reward  $\leftarrow$  1
9:     Update q_value positively
10:  end if
11:  for each param in selected_params do
12:    for each param_name, param_value in param.items() do
13:      old_q_value  $\leftarrow$  q_table[operation_id][param_name]
14:      max_q_value_next_state  $\leftarrow$  max(q_table[operation_id].values())
15:      q_table[operation_id][param_name]  $\leftarrow$  old_q_value +  $\alpha$  * (reward +  $\gamma$  * (max_q_value_next_state - old_q_value))
16:    end for
17:  end for
18:  return q_table, q_value
19: end procedure
1: procedure MAIN(API specification)
2:   Initialize  $\epsilon_{max} \leftarrow 1$ 
3:   Initialize  $\epsilon_{adapt} \leftarrow 1.1$ 
4:   Initialize time_limit  $\leftarrow$  desired time limit in seconds
5:   operations  $\leftarrow$  Load API specification
6:    $\alpha$ ,  $\gamma$ ,  $\epsilon$ , q_table, q_value  $\leftarrow$  INITIALIZEQLEARNING(operations)
7:   while time_limit not reached do
8:     operation  $\leftarrow$  SELECTOPERATION(operations, q_table)
9:     parameters  $\leftarrow$  SELECTPARAMETERS(operation,  $\epsilon$ )
10:    source  $\leftarrow$  SELECTVALUEMAPPINGSOURCE(operation,  $\epsilon$ )
11:    response  $\leftarrow$  Execute API request with operation, parameters, and source
12:    q_table, q_value  $\leftarrow$  QLEARNINGUPDATER(response, q_table, q_value, operation, parameters,  $\alpha$ ,  $\gamma$ )
13:     $\epsilon \leftarrow \min(\epsilon_{max}, \epsilon_{adapt} * \epsilon)$ 
14:  end while
15: end procedure
```

OpenAPI documentation [6] states that users can specify example values in the *description* field, and a recent study also shows the importance of leveraging example values from descriptions [144]. However, such examples are not provided in a structured format but as natural-language text. To extract example values from the *description* field, I create a list containing each word in the text, as well as each quoted phrase.

- Source 2 (random value generated based on parameter’s type, format, and constraints): This source generates random values for each parameter based on its type, format, and constraints. To generate random values, I utilize Python’s built-in *random* library [154]. For date and date-time formats, I employ the *datetime* library [155] to randomly select dates and times. If the parameter has a regular ex-

pression pattern specified in the API documentation, I generate the value randomly using the *rstr* library [156]. When a minimum or maximum constraint is present, I pass it to the *random* library to ensure that the generated values adhere to the specified constraints. This approach allows ARAT-RL to explore a broader range of values compared to the example values provided in the API specification.

- Source 3 (dynamically constructed key-value pairs from request): This source extracts key-value pairs from dynamically constructed request key-value pairs. I employ Gestalt pattern matching [157] (also known as Ratcliff/Obershelp pattern recognition [158]) to identify the key most similar to the parameter name. Gestalt matching is a lightweight but effective technique that calculates the similarity between two strings by identifying the longest common subsequences and recursively comparing the remaining unmatched substrings. This technique aids in discovering producer-consumer relationships.
- Source 4 (dynamically constructed key-value pairs from response): Similar to Source 3, this source obtains key-value pairs from dynamically constructed response key-value pairs. I use the same Gestalt pattern matching approach [157] to identify the key, further assisting in the identification of producer-consumer relationships.
- Source 5 (default values): This source uses predefined default values for each data type: “string” for strings, 1.1 for numbers, 1 for integers: 1, [] for arrays, and {} for objects. These default values can be useful for testing how the API behaves when it receives the simplest or common forms of inputs; such default values are used by other tools as well (e.g., [9]).

Similar to the *SelectParameters* procedure, the selection of the value-mapping source is guided by ϵ . If a random value is greater than ϵ , the algorithm selects the mapping source with the highest Q-value for the chosen operation (lines 4–6). This helps the algorithm focus on the most-promising mapping sources based on prior experience. Otherwise, the

algorithm randomly selects a mapping source from the available sources (line 8). This randomness ensures that the algorithm occasionally explores less-promising mapping sources to avoid getting stuck in a suboptimal solution.

6.3.3 Q-Learning-based API Testing

In this step, ARAT-RL executes the selected API operations with the selected parameters and value-mapping sources, and updates the Q-values based on the response status codes. Algorithm Algorithm 6 (*QLearningUpdater* and *Main*) describes the API testing process and the update of Q-values using the learning rate (α) and discount factor (γ).

The *QLearningUpdater* procedure updates the Q-values based on the response status codes. It first extracts the operation ID from the selected operation (line 2). If the response status code indicates success (2xx), the algorithm extracts key-value pairs from the request and response, assigns a reward of -1 , and updates the Q-values negatively (lines 3–6). If the response status code indicates an unsuccessful request (4xx or 500), the algorithm assigns a reward of 1 and updates the Q-values positively (lines 7–10). The Q-values are updated for each parameter in the selected parameters using the Q-learning update rule (Equation Equation 2.1 in §subsection 2.1.5 (lines 11–16), and the updated Q-values are returned (line 18).

The *Main* procedure orchestrates the Q-Learning-based API testing process. It initializes the exploration rate (ϵ), its maximum value (ϵ_{max}), its adaptation factor (ϵ_{adapt}), and the time budget for testing (lines 2–4). The API specification is loaded and the Q-Learning table is initialized (lines 5–6). The algorithm then enters a loop that continues until the time limit is reached (line 7). In each iteration, the best operation, selected parameters, and selected mapping source are determined (lines 8–10). The API operation is executed with the selected parameters and mapping source, and the response is obtained (line 11). The Q-values are then updated based on the response (line 12), and the exploration rate (ϵ) is updated (line 13).

By continuously updating the Q-values based on the response status codes and adapting the exploration rate, Q-Learning-based API testing process aims to effectively explore the space of API operations and parameters.

6.4 Evaluation

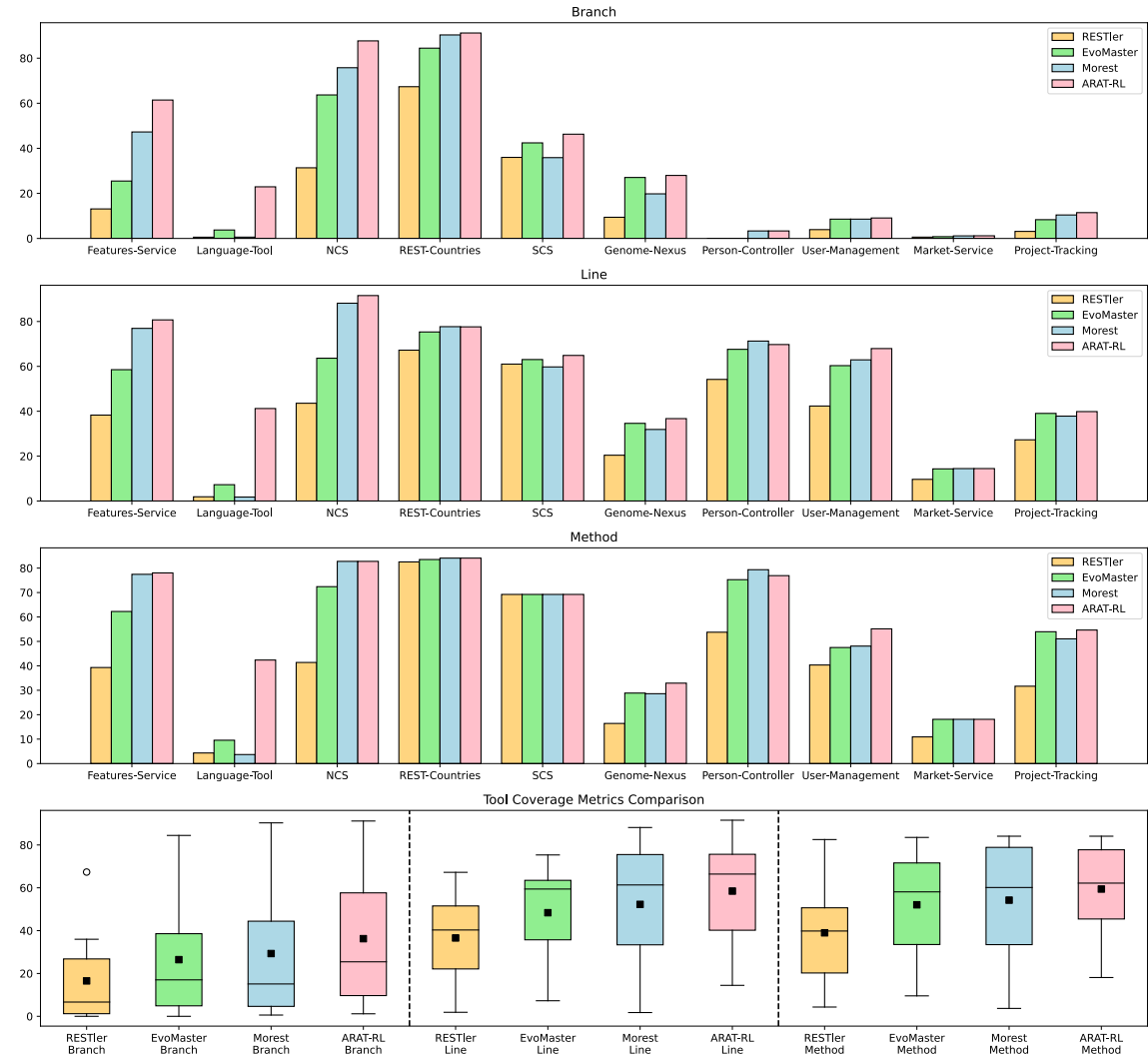


Figure 6.3: Branch, line, and method coverage achieved by the tools on the benchmark services.

In this section, I present the results of empirical studies conducted to assess ARAT-RL. This evaluation aims to address the following research questions:

1. **RQ1:** How does ARAT-RL compare with state-of-the-art REST API testing tools in

terms of code coverage?

2. **RQ2:** How does the efficiency of ARAT-RL, measured in terms of valid and fault-inducing requests generated and operations covered within a given time budget, compared to that of other REST API testing tools?
3. **RQ3:** In terms of error detection, how does ARAT-RL perform in identifying 500 responses in REST APIs compared to state-of-the-art REST API testing tools?
4. **RQ4:** How do the main components of ARAT-RL—prioritization, dynamic key-value pair construction, and sampling—contribute to its overall performance?

6.4.1 Experiment Setup

I performed the experiments using Google Cloud E2 machines, each equipped with 24-core CPU and 96 GB of RAM. I created a machine image containing all the services and tools in this benchmark. For each experiment, I deleted and recreated the machines using this image to minimize potential dependency issues. Each machine hosted all the services and tools under test, but I ran one tool at a time during the experiments. I monitored CPU and memory usage throughout the testing process to ensure that the testing tools were not affected by a lack of memory or CPU resources.

To evaluate the effectiveness and efficiency of this approach, I compared its against three state-of-the-art tools: EvoMaster [45], RESTler [9], and Morest [17]. I selected 10 RESTful services from a recent REST API testing study [19] as the benchmark. I explain the selection process of these tools and services next.

Testing Tools Selection

As a preliminary note, because ARAT-RL is a black-box approach, I considered black-box tools only in this evaluation. I believe that adding white-box tools to the evaluation would

result in an unfair comparison, as these tools leverage information about the code, rather than just information in the specification, to generate test inputs.

I identified an initial set of 10 tools based on my previous study [19]. From this list, I chose (the black-box version of) EvoMaster [45] and RESTler [9]. EvoMaster employs an evolutionary algorithm for automated test case generation and was the best-performing tool in that study in another recent comparison [20]. Its strong performance makes it an appropriate candidate for comparison. RESTler adopts a grammar-based fuzzing approach to explore APIs. It is a well-established tool in the field and, in fact, the most popular REST API testing tool in terms of GitHub stars.

Recently, two new tools have been published. Morest [17] has been shown to have superior results compared to EvoMaster. I, therefore, included Morest in the set of tools for comparison, as it could potentially outperform the other tools. The other recent tool, RestCT, was also considered for inclusion in this evaluation. However, I encountered failures while running it. I contacted the authors, who confirmed the issues and said they will work on resolving them.

RESTful Services Selection

As benchmarks for the evaluation, I selected 10 out of 20 RESTful services from my previous study [19]. I had to exclude 10 services for various reasons. Specifically, I omitted the News service, developed by the author of one of the baseline tools (EvoMaster), to avoid possible bias. Problem Controller and Spring Batch REST were excluded because they require specific domain knowledge to generate meaningful tests, so using them provides limited information about the tools. I excluded Erc20 Rest Service and Spring Boot Actuator because some APIs in these services did not provide valid responses due to external dependencies being updated without corresponding updates in the service code. Proxyprint, OCVN, and Scout API could not be included due to authentication issues that prevented them from generating meaningful responses. Finally, I excluded CatWatch and CWA Veri-

Table 6.1: Comparison of operations covered and valid and failure-inducing requests generated (2xx and 500 status codes) by the tools.

Service	ARAT-RL				Morest			EvoMaster			RESTler					
	#operations covered	#requests		500	#operations covered	#requests		500	#operations covered	#requests		500	#operations covered	#requests		500
Features Service	18	95,479	43,460	52,019	18	103,475	4,920	98,555	18	113,136	33,271	79,865	17	4,671	1,820	2,851
Language Tool	2	77,221	67,681	9,540	1	1,273	1,273	0	2	22,006	17,838	4,168	1	32,796	32,796	0
NCS	6	62,618	62,618	0	5	18,389	18,389	0	2	61,282	61,282	0	2	140	140	0
REST Countries	22	36,297	35,486	811	22	8,431	7,810	621	16	9,842	9,658	184	6	259	255	4
SCS	11	115,328	115,328	0	11	110,147	110,147	0	10	66,313	66,313	0	10	5,858	5,857	1
Genome Nexus	23	15,819	14,010	1,809	23	32,598	10,661	21,937	19	8,374	8,374	0	1	182	182	0
Person Controller	12	101,083	47,737	53,346	11	104,226	10,036	94,190	12	91,316	37,544	53,772	1	167	102	65
User Management Microservice	21	44,121	13,356	30,765	17	1,111	948	163	18	29,064	13,003	16,061	4	79	64	15
Market Service	12	29,393	6,295	23,098	6	1,399	394	1,005	5	10,697	4,302	6,395	2	1,278	0	1,278
Project Tracking System	53	23,958	21,858	2,100	42	14,906	12,904	2,002	43	15,073	13,470	1,603	3	72	65	7
Average	18	60,132	42,783	17,349	15.6	39,595	17,748	21,847	14.5	42,710	26,505	16,205	4.7	4,550	4,128	422

fication because of their restrictive rate limits, which slowed down the testing process and made it impossible to collect results in a reasonable amount of time.

The final set of services consisted of Features Service, LanguageTool, NCS, REST Countries, SCS, Genome Nexus, Person Controller, User Management Microservice, Market Service, and Project Tracking System.

Result Collection

I ran each testing tool with the time budget of one hour per execution, as my previous study [19] showed that code coverage achieved by these tools tends to plateau within this duration. To accommodate randomness, I replicated the experiments ten times and calculated the average metrics across the runs.

Data collection for code coverage and status codes was done using JaCoCo [114] and Mitmproxy [159], respectively. I focused on identifying unique instances of 500 codes, which indicate server-side faults. The methodology was as follows:

1. **Stack Trace Collection:** For services that provided stack traces with 500 errors, I collected these traces, treating each unique trace as a separate fault. In the majority of cases, the errors I collected fall into this category.
2. **Response Text Analysis:** In the absence of stack traces, I analyzed the response text. After removing unrelated components (e.g., timestamps), I classified unique instances of response text linked to 500 status codes as individual faults.

This systematic approach allowed me to compile a comprehensive and unique tally of faults for this analysis.

6.4.2 RQ1: Effectiveness

To answer RQ1, I compared the tools in terms of branch, line, and method coverage achieved. Figure 6.3 presents the results of the study. The bar charts represent the coverage achieved by each tool for each RESTful service, whereas the boxplot at the bottom summarizes of each tool’s performance on the three coverage metrics across all the services.

As the boxplot illustrates, ARAT-RL consistently outperforms the other tools in all three coverage metrics over the subject services. Looking at the performance breakdown by services (shown in the bar charts), ARAT-RL performed the best (or was tied as the best-performing tool) for all services in terms of branch coverage (with one tie), eight of the 10 services in terms of line coverage (with no tie), and nine of the 10 services in terms of method coverage (with four ties). In cases where ARAT-RL did not achieve the best results, it still achieved similar coverage rates as the best-performing tool.

ARAT-RL is especially effective when there are operation dependencies, parameter dependencies, and value-mapping dependencies. For example, the highest coverage gains occurred for Language Tool, which has a complex set of inter-parameter and value-mapping dependencies. However, ARAT-RL struggles with semantic parameters—parameters whose valid values must adhere to real-world constraints and specific contextual meanings. These parameters require values that are not only syntactically correct but also semantically valid (e.g., addresses, emails, names, passwords, and phone numbers). For instance, ARAT-RL’s effectiveness was the lowest on Market Service, although it still matched the best-performing tool on this service. The reason for this is that the service requires input data such as address, email, name, password, and phone number in specific formats, but ARAT-RL failed to generate valid values for these. Consequently, it was unable to cre-

ate market users and then use that information for other operations in producer-consumer relationships.

On average, ARAT-RL attained 36.25% branch coverage, 58.47% line coverage, and 59.42% method coverage. In comparison, Morest, which exhibited the second-best performance, reached an average of 29.31% branch coverage, 52.27% line coverage, and 54.24% method coverage. Thus, the coverage gain of ARAT-RL over Morest is 23.69% for branch coverage, 11.87% for line coverage, and 9.55% for method coverage. EvoMaster and RESTler yield lower average coverage rates on all three metrics, with respective results of 26.45%, 48.37%, and 52.07% for EvoMaster and 16.54%, 36.58%, and 38.99% for RESTler for branch, line, and method coverage. The coverage gains of ARAT-RL compared to EvoMaster is 37.03% for branch coverage, 20.87% for line coverage, and 14.13% for method coverage; compared to RESTler, the gains are 119.17% for branch coverage, 59.83% for line coverage, and 52.42% for method coverage.

These results provide evidence that this technique can more effectively explore REST services, achieving superior code coverage, compared to existing tools, and demonstrate its potential in addressing the challenges in REST API testing.

ARAT-RL consistently outperforms RESTler, EvoMaster, and Morest in terms of branch, line, and method coverage across the subject services. However, ARAT-RL can struggle with parameters that require inputs in specific formats.

6.4.3 RQ2: Efficiency

To address RQ2, I compared ARAT-RL to Morest, EvoMaster, and RESTler in terms of the number of (1) valid and fault-inducing requests generated (as indicated by HTTP status codes 2xx and 500, respectively) and (2) operations covered within a given time budget. Although efficiency is not only dependent on these metrics, due to factors such as API response time, I feel that they still represent meaningful proxies because they indicate the

Table 6.2: Total faults detected by the tools over 10 runs.

Service	RESTler	EvoMaster	Mostest	ARAT-RL
Features Service	10	10	10	10
Language Tool	0	48	0	122
NCS	0	0	0	0
REST Countries	9	10	10	10
SCS	3	0	0	0
Genome Nexus	0	0	5	10
Person Controller	58	221	274	943
User Management Microservice	10	10	8	10
Market Service	10	10	10	10
Project Tracking System	10	10	10	10
Total	110	319	327	1125

extent to which the tools are exploring the API and identifying faults.

Table 6.1 shows these metrics for the 10 subject services. For each service, the table lists the number of operations covered and the number of requests made under the categories 2xx+500 (sum of 2xx code and 500 status code), 2xx, and 500, for each of the four tools. In the last row, the table presents the average number of operations covered and requests made by each tool across all the services.

In the testing time budget of one hour, ARAT-RL generated more valid and failure-inducing requests, resulting in more exploration of the testing search space. Specifically, ARAT-RL generated 60,132 valid and fault-inducing requests on average, which is 52.01% more than Mostest (39,595 requests), 40.79% more than EvoMaster (42,710 requests), and 1222% more than RESTler (4,550 requests).

This difference in the number of requests can be attributed to ARAT-RL’s approach of processing only a sample of key-value pairs from the response, rather than the entire response. By focusing on sampling key-value pairs, ARAT-RL efficiently identifies potential areas of improvement, contributing to a more effective REST API testing process.

Moreover, ARAT-RL covered more operations on average (18 operations) compared to Mostest (15.6 operations), EvoMaster (14.5 operations), and RESTler (4.7 operations). This indicates that ARAT-RL efficiently generates more requests in a given time budget, which leads to covering more API operations, contributing to more comprehensive testing

process.

Given a one-hour testing time budget, ARAT-RL generates 52.01%, 40.79%, and 1222% more valid and fault-inducing requests than Morest, EvoMaster, and RESTler, respectively. Additionally, it covers 15.38%, 24.14%, and 282.98% more operations than these tools.

6.4.4 RQ3: Fault-Detection Capability

Table 6.2 presents the total number of faults detected by each tool across 10 runs for the RESTful services in this benchmark. I note that this cumulative count might include multiple detection of the same fault in different runs. For clarity in discussion, I provide the average faults detected per run in the text below. As indicated, ARAT-RL exhibits the best fault-detection capability, uncovering on average 113 faults over the services. In comparison, Morest and EvoMaster detected 33 and 32 faults on average, respectively, whereas RESTler found the least—11 faults on average. ARAT-RL thus uncovered 9.3x, 2.5x, and 2.4x more faults than RESTler, EvoMaster, and Morest, respectively.

Comparing this data against the data on 500 response codes from Table 6.1, I can see that, although ARAT-RL generated 20.59% fewer 500 responses, it found 250% more faults than Morest. Compared to EvoMaster, ARAT-RL generated 7.06% more 500 responses, but also detected 240% more faults. These results suggest that irrespective of whether ARAT-RL generates more or fewer error responses, it is better at uncovering more unique faults.

ARAT-RL’s superior fault-detection capability is particularly evident in the Language Tool and Person Controller services, where it detected on average 12 and 94 faults, respectively. What makes this significant is that these services have larger sets of parameters. This strategy performed well in efficiently prioritizing and testing various combinations of these parameters to reveal faults. For example, Language Tool’s main operation `/check`, which checks text grammar, has 11 parameters. Similarly, Person Controller’s main operation `/api/person`, which creates/modifies person instances, has eight parameters. In

Table 6.3: Results of the Ablation Study.

	Branch	Line	Method	Faults Detected
ARAT-RL	36.25	58.47	59.42	112.10
ARAT-RL (no prioritization)	28.70 (+26.3%)	53.27 (+9.8%)	55.51 (+7%)	100.10 (+12%)
ARAT-RL (no feedback)	32.69 (+10.9%)	54.80 (+6.9%)	56.09 (+5.9%)	110.80 (+1.2%)
ARAT-RL (no sampling)	34.10 (+6.3%)	56.39 (+3.7%)	57.20 (+3.9%)	112.50 (-0.4%)

contrast, the other services’ operations usually have three or fewer parameters.

ARAT-RL intelligently tries various parameter combinations with a reward system in Q-learning because it gives negative rewards for the parameters in the successful requests. This ability to explore various parameter combinations is a significant factor in revealing more bugs, especially in services with a larger number of parameters. These results indicate that ARAT-RL’s reinforcement-learning-based approach can effectively discover faults in REST services.

ARAT-RL exhibits superior fault-detection ability, uncovering 9.3x, 2.5x, and 2.4x more bugs than RESTler, EvoMaster, and Morest, respectively. This is mainly attributed to its intelligent RL-based exploration of various parameter combinations in services with large number of parameters.

6.4.5 RQ4: Ablation Study

To address RQ4, I conducted an ablation study to assess the impact of the main novel components of ARAT-RL on its performance. I compared the performance of ARAT-RL to three variants: (1) ARAT-RL without prioritization, (2) ARAT-RL without dynamic key-value construction from feedback, and (3) ARAT-RL without sampling. Table 6.3 presents the results of this study.

As illustrated in the table, the removal of any component results in reductions in branch, line, and method coverage, as well as the number of faults detected. Eliminating the prioritization component leads to the most substantial decline in performance, with branch,

line, and method coverage decreasing to 28.70%, 53.27%, and 55.51%, respectively, and the number of detected faults dropping to 100. This highlights the critical role of the prioritization mechanism in ARAT-RL's effectiveness.

The absence of feedback and sampling components also negatively affects performance. Without feedback, ARAT-RL's branch, line, and method coverage decreases to 32.69%, 54.80%, and 56.09%, respectively, and the number of found faults is reduced to 110.80. Likewise, without sampling, branch, line, and method coverage drops to 34.10%, 56.39%, and 57.20%, respectively, while the number of found faults experiences a slight increase to 112.50, which may be attributed to random variations.

These findings emphasize that each component of ARAT-RL is essential for its overall effectiveness. The prioritization mechanism, feedback loop, and sampling strategy work together to optimize the tool's performance in terms of code coverage and fault detection capabilities.

Each component of ARAT-RL—prioritization, feedback, and sampling—contributes to ARAT-RL's overall effectiveness. The prioritization mechanism, in particular, plays a significant role in enhancing ARAT-RL's performance in code coverage achieved and faults detected.

6.4.6 Threats to Validity

In this section, I address potential threats to this study's validity and the steps taken to mitigate them. Internal validity is influenced by tool implementations and configurations. To minimize these threats, I used the latest tool versions and used default options. Some testing tools might have randomness, but I addressed this issue by running each tool 10 times and computing the average results.

The data-collection process relies on an automated script. Despite thorough testing, there remains the possibility that the script could contain errors, potentially affecting the results. To mitigate this risk, I conducted several rounds of pilot testing on a representative

subset of the services before applying the script to the full study.

External validity is affected by the selection of RESTful services and the limited number of RESTful services tested, impacting the generalizability of the findings. I tried to ensure a fair evaluation by selecting a diverse set of 10 services, but I will try to have a larger and more diverse set in future work.

Construct validity concerns the metrics and tool comparisons used. Metrics such as branch, line, and method coverage, number of requests, and 500 status codes as faults, although commonly used, may not capture the test tools' full quality. Additional metrics, such as mutation scores, could provide a better understanding of tool effectiveness. I measured efficiency by considering the number of meaningful requests generated by the tools, including valid and fault-inducing ones, as well as the number of operations each tool covered within a given time budget. While these metrics give me some perspective on a tool's performance, they do not consider all possible factors. Other aspects, such as the response times from the services, may also significantly impact overall efficiency.

Including more tools, services, and metrics in future studies would allow a more comprehensive evaluation of this technique's performance.

6.5 Conclusion

I introduced ARAT-RL, a reinforcement-learning-based approach for the automated testing of REST APIs. To assess the effectiveness, efficiency, and fault-detection capability of this approach, I compared its performance to that of three state-of-the-art testing tools. The results show that ARAT-RL outperformed all three tools considered in terms of branch, line, and method coverage achieved, requests generated, and faults detected. I also conducted an ablation study, which highlighted the important role played by the novel features of this technique—prioritization, dynamic key-value construction based on response data, and sampling from response data to speed up key-value construction—in enhancing ARAT-RL's performance.

CHAPTER 7

EFFECTIVE REST API TESTING WITH SMALL LANGUAGE MODELS

7.1 Introduction

Despite advancements in REST API testing, significant challenges remain due to complex input constraints and inter-parameter dependencies among API parameters of real-world RESTful services [19, 106]. To address these challenges, researchers have turned to recent advances in the field of natural language processing (NLP), leading to several promising approaches for REST API testing. For example, NLP2REST [144], my previous work, employs NLP techniques to extract constraints and dependencies from OAS and convert them into OAS keywords (e.g., `minimum`, `maximum`, `pattern`) that testing tools can use. ARTE [50] generates realistic parameter inputs by extracting key information from parameter names and descriptions and querying knowledge bases, such as DBPedia [51]. RESTGPT [160] uses Large Language Models (LLMs) to perform rule extraction and parameter input generation. These tools have improved REST API testing effectiveness by extracting additional machine-readable (structured) information from the human-readable portions of API specifications. However, they lack the ability to dynamically interact with testing tools during execution. This limitation prevents them from refining test inputs and rules based on real-time server feedback.

To address these limitations, I present LlamaRestTest, the first black-box testing approach that leverages Language Models (LMs) to incorporate dynamic server feedback during REST API testing. LlamaRestTest consists of two specialized LMs created by fine-tuning the Llama3-8b model [161]: LlamaREST-IPD for identifying inter-parameter dependencies, and LlamaREST-EX for generating appropriate input values. These models were fine-tuned using a comprehensive dataset combining Martin-Lopez’s dependency

database [106] and over 1.8 million API operation parameters mined from APIs-guru [162]. I also created quantized versions of these models to investigate their efficiency-accuracy tradeoff when using lower precision. LlamaRestTest further leverages these models together with reinforcement learning for more effective API exploration. Specifically, I integrated these models with ARAT-RL [15]¹, a reinforcement learning-based testing approach, enabling dynamic analysis of parameter descriptions and server responses to generate valid parameter combinations and input values during testing.

To assess the performance of LlamaRestTest, I collected 12 real-world RESTful services, previously evaluated in [144, 15], including popular services such as Spotify. I compared the tool with RESTGPT to evaluate the effectiveness of LlamaREST-IPD and LlamaREST-EX in identifying inter-parameter dependencies and computing valid input values, respectively, using various configurations, including the base Llama3-8B model, the fine-tuned model, and the fine-tuned model with 2-bit, 4-bit, and 8-bit quantization. Additionally, I investigated how fine-tuning and quantization impact overall testing performance. I then compared LlamaRestTest with four state-of-the-art REST testing tools: RESTler [9], EvoMaster [45], MoRest [17], and ARAT-RL [15]. I measured effectiveness using three key metrics widely adopted in API testing research [163, 19]: code coverage (for open-source services), operation coverage (for online services), and the ability to detect internal server errors (for all services). Finally, I conducted an ablation study to assess how specification information, server response messages, parameter input generation, and parameter dependency identification each contribute to LlamaRestTest’s overall performance.

The empirical evaluation demonstrated LlamaRestTest’s effectiveness across multiple dimensions. In parameter handling, the fine-tuned LlamaREST-EX model achieved a 72.44% success rate in generating valid inputs, outperforming the base Llama3-8B model (22.94%) while also being more effective than RESTGPT (68.82%). For inter-parameter dependencies, LlamaREST-IPD successfully identified 12 out of the 17 dependencies,

¹I chose ARAT-RL as its exploration strategy has been shown to be effective among the recent tools [15].

whereas RESTGPT and the base Llama model identified nine and two dependencies, respectively. In terms of code coverage, the fine-tuned model with 8-bit quantization achieved the best balance, reaching the highest average coverage (55.8% method, 28.3% branch, and 55.3% line). This model processed 37.5 operations per run—14.5 more operations processed per run than the base Llama model (which processed 23 operations). Fine-tuning alone provided modest benefits (50.1% method, 23.5% branch, 49.9% line coverage, and 24.5 operations processed), showing improvements of 0.9 percentage points, 0.3 percentage points, 1.4 percentage points, and 1.5 additional operations respectively over the base model (49.2%, 23.2%, 48.5%, and 23 operations). The 4-bit quantized model maintained strong performance (54.0% method, 27.9% branch, 53.5% line coverage, and 34.4 operations processed), whereas the 2-bit model showed lower but still meaningful improvements (52.9% method, 25.7% branch, 52.6% line coverage, and 25.4 operations processed) compared to both the base and the fine-tuned model, highlighting the benefit of quantization in REST API testing.

When compared against state-of-the-art testing tools (EvoMaster, ARAT-RL, RESTler, and MoRest), LlamaRestTest demonstrated substantial improvements in coverage (9.2–18.7% more branch coverage, 10.0–22.1% more line coverage, and 10.0–22.7% more method coverage) while identifying more internal server errors (204 compared to 130–160). The ablation study revealed that removing LlamaREST-IPD had the most significant impact (reducing coverage by 6.2–9.2%), followed by server response analysis (4.4–6.3% reduction), parameter descriptions (1.4–5.9% reduction), and LlamaREST-EX (3.0–4.5% reduction).

The main contributions of this work are:

- LlamaRestTest, the first LM-based REST API testing approach that leverages server feedback along with fine-tuning and quantization techniques to improve both effectiveness and efficiency in REST API testing.
- Empirical results that demonstrate the effectiveness of fine-tuning and quantization

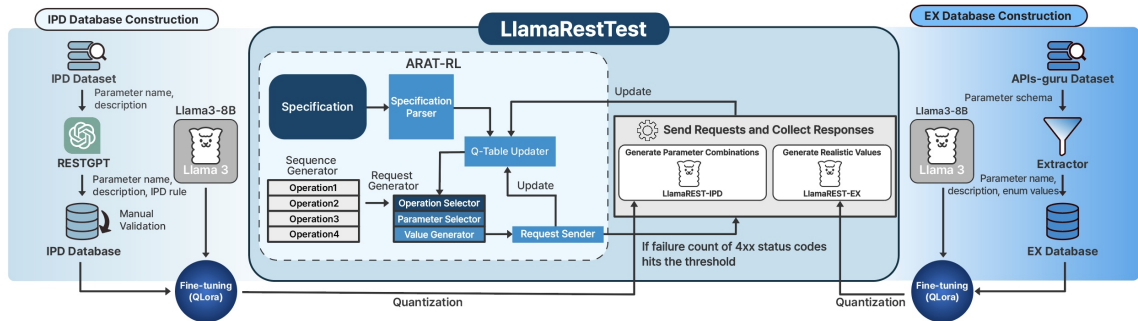


Figure 7.1: Overview of the approach.

in REST API testing and show that LlamaRestTest outperforms the current state-of-the-art testing tools.

- An artifact [164] that includes the LlamaRestTest tool, benchmarking services, and the data used for fine-tuning, along with empirical study results.

7.2 Approach

Figure 7.1 presents an overview of LlamaRestTest. LlamaRestTest is designed to address the challenges of REST API testing, particularly in handling complex IPDs and generating valid inputs. In this section, I discuss the key features of LlamaRestTest, including model selection, database construction, fine-tuning, and quantization, which together enable adaptive, efficient, and effective testing. These components work in synergy to generate and refine API requests dynamically based on server feedback, improving both test coverage and fault detection.

7.2.1 Base Model Selection

LlamaRestTest uses LLMs to identify IPDs and generate valid values for API parameters. The closest approach is RESTGPT which leverages OpenAI’s ChatGPT-3.5 Turbo model to enhance OpenAPI specifications with rules for REST API testing, performing similar tasks—i.e., identifying IPDs and generating example values. However, the cost—approximately \$10 per specification—poses a challenge, especially when considering the

large number of LLM interactions associated with dynamic server interactions. Moreover, RESTGPT focuses solely on static specifications, whereas this approach seeks to leverage server response messages at runtime. Given also the potential exposure of sensitive information in REST API testing scenarios, I prioritized using open-source models that can be hosted on the user’s computing resource.

This work’s goal was to employ powerful yet lightweight LLMs to reduce costs, improve speed, and ensure compatibility with various computing environments, including CPU-based systems. However, I encountered challenges with LLMs, such as the Llama3-8B model, which still requires considerable GPU resources [165]. Additionally, Llama3-8b lacked the capability to generate valid IPD rules and example values tailored for REST APIs when tested using prompts from RESTGPT, as shown by the results (Table 7.2 and Table 7.3).

To address these issues, I applied fine-tuning to enhance the model’s capabilities for REST API testing and employed quantization techniques to make the model more lightweight. Another reason for selecting Llama3-8B as the baseline model is its architecture’s support for various quantization options, including popular 2-bit, 4-bit, and 8-bit integer formats [166, 167, 168]. Llama3 is also open-source and offers a GPT-Generated Unified Format (GGUF) transformation option—a binary format designed for fast model loading and saving.

7.2.2 Database Construction

Figure 7.1 illustrates how I constructed the fine-tuning databases. I systematically created two databases: the IPD database and the example (EX) database. The goal of this process was to collect parameter names, descriptions, and associated rules for the IPD database, which includes the Requires, Or, OnlyOne, AllOrNone, ZeroOrOne, and Arithmetic/Relational/Complex IPD rules [106]. For the EX database, I collected parameter names, descriptions, and semantically valid values.

For mining IPD data, I used Martin-Lopez’s IPD database [106], which contains manually collected dependencies from REST APIs. To ensure a fair evaluation on the benchmark, I excluded any OpenAPI specifications that were included in the benchmark dataset. After filtering, I were left with 551 parameters with IPDs. For creating the EX database, I leveraged APIs-guru [162], which provides a large collection of OAS. I gathered 4,133 OpenAPI specifications from this dataset, excluding those used in the benchmark. These specifications include over 1.8 million parameter sets for REST APIs. I specifically targeted parameters with enumerated values which are predefined set of valid values (e.g., `json` and `xml` for a parameter on data format).

To generate IPD rules for the parameters in the IPD database, I used RESTGPT [160]. RESTGPT extracts IPD rules from natural language descriptions in API specifications. After extracting IPD rules automatically using RESTGPT, I manually validated the rules. The manual validation involved checking whether the extracted IPD rules accurately represented the intended parameter dependencies based on their descriptions. Out of 551 IPDs, 25 (4.5%) required manual fixing. These were primarily complex cases involving more than two parameters or compound rules. I performed the validation. The process took approximately five hours in total.

Rules were validated based on their logical consistency with the parameter descriptions. For example, a rule initially extracted as “Requires(A, B) AND Requires(B, C)” might be corrected to “AllOrNone(A, B, C)” after manual validation. Similarly, the rule “OnlyOne(X, Y) OR ZeroOrOne(Z)” might be modified to “OnlyOne(X, Y, Z)” to reflect the intended behavior more accurately. It is important to note that manual validation is a one-time step per API version. The validation step is optional and should have limited effect on the results, especially for the dataset where only 4.5% of IPDs required correction. To support reproducibility, I have shared the models and the validated training dataset in the artifact [169].

For the EX database, I focused on extracting parameters with enumerated values. From

the API documentation, I systematically extracted the *parameter name*, the corresponding *description*, and the specified *enum values* to create the training dataset. This data was structured such that the parameter name and description served as inputs, whereas the valid enum values were used as outputs for model fine-tuning.

The IPD and EX databases share a common schema to ensure compatibility and ease of use during the fine-tuning process. Each database entry includes the following fields:

- **Parameter Name:** Specifies the name of the API parameter (e.g., `bboxes`, `filter`, `format`).
- **Description:** A brief description of the parameter’s purpose, extracted from the OAS or generated by RESTGPT.
- **Inter-Parameter Dependency (IPD):** For the IPD database, this field outlines the dependency rules associated with the parameter (e.g., mutually exclusive parameters or required groups of parameters). These rules were generated by RESTGPT and manually validated.
- **Example Values (EX):** For the EX database, this field contains an array of valid enum values for the parameter, as specified in the OpenAPI documentation.

7.2.3 Fine-Tuning and Quantization

Dataset Preparation

The fine-tuning datasets for IPD detection and example value generation were preprocessed and reformatted into a custom tokenized format suitable for fine-tuning. Initially, I attempted to fine-tune a single model for both tasks. However, the preliminary investigations revealed that this approach led to a poor performance in IPD detection, with the model displaying catastrophic forgetting issues. This led me to the decision of creating separate fine-tuned models to improve overall performance.

For each parameter, I used a structured format with special tokens to provide context and guide the model’s training. The LlamaREST-IPD model’s input sequences follow this structure:

```
<s>[INST] Find Inter-Parameter Dependency for the parameter below
name: travelMode
description: If startTime is present, travelMode must be 'driving'. [/INST]
IF startTime THEN travelMode == 'driving';</s>
```

This format includes the parameter’s name and description, followed by the dependency rule in a structured manner, enclosed with special tokens `<s>` and `[INST]`. Here, `<s>` and `</s>` mark the start and end of the prompt, respectively. Similarly, `[INST]` and `[/INST]` indicate the start and end of the instruction. The text between `<s>[INST]` and `[/INST]` is the instruction, which provides the context; the text after `[/INST]` presents the expected rule. This setup allows the model to learn both the natural language description and the corresponding logic for IPDs.

For LlamaREST-EX, the data is formatted to extract and list valid example values for each parameter. An example of this structure is:

```
<s>[INST] Find example values for the parameter below in a list format
name: Content-Type
description: The content type. [/INST]
['application/x-www-form-urlencoded', 'application/json', ...]</s>
```

This tokenization ensures that the model understands the task of generating example values for each parameter in list format. The input includes the parameter name and description, while the output provides a list of valid example values. I experimented with different tokenization schemes and ultimately chose Llama2 tokens, as Llama3’s agentic token support is not required for REST API testing tasks, and Llama2 tokens are simpler. The structured format and custom tokens facilitate efficient parsing of IPDs and example values during fine-tuning.

Parameter-Efficient Fine-Tuning

Fine-tuning plays a pivotal role in enhancing the performance of LlamaREST-IPD and LlamaREST-EX by enabling the models to adapt to the tasks of IPD detection and example value generation. Fine-tuning can be performed in two ways: Full Parameter Fine-Tuning (FFT) and Parameter-Efficient Fine-Tuning (PEFT). FFT requires large datasets and extensive computational resources, whereas PEFT is designed to work efficiently with smaller datasets, making it a better fit for the application. IPD dataset contains only 551 parameters, which is insufficient for FFT, as prior research has shown that fewer than 1,000 examples do not yield significant performance improvements with FFT [170]. I therefore employed PEFT, which optimizes the model by fine-tuning only a subset of its parameters. This selective updating process allows it to efficiently adapt to new tasks using smaller datasets and fewer computational resources.

The two most commonly used PEFT methods are LoRA [35] and QLoRA [36]. Both approaches allow fine-tuning with significantly reduced computational requirements. LoRA focuses on injecting trainable low-rank matrices into transformer layers, whereas QLoRA quantizes model weights during training, further reducing memory usage and computational costs. Given the limitations of the compute environment, which includes a Google Colab machine with a 40GB A100 GPU, I opted for QLoRA. QLoRA's quantization compresses the model, letting it fit within memory constraints while still performing the necessary updates.

I based the fine-tuning configuration on the recommended settings from Meta's official guidelines [171], making necessary adjustments to accommodate the hardware. Specifically, I reduced the batch size to 8 to fit within the 40GB VRAM of the A100 GPU. As mentioned, the fine-tuning process leverages a structured dataset specifically designed for IPD detection and value generation tasks, using custom tokens to represent the relationships between parameters. These tokens guide the model's understanding of the dependencies and example values, allowing for updates of only the relevant parts of the model during

fine-tuning. This adaptation allows for faster convergence and lower computational costs compared to full parameter tuning.

To prevent overfitting and reduce unnecessary computation, I applied an early stopping strategy [172]. Initially, I set the epoch limit to 100 but continuously monitored the training loss. Early stopping was triggered when the loss plateaued—meaning further training would no longer improve the model’s performance. This resulted in 35 epochs for LlamaREST-EX and 51 epochs for LlamaREST-IPD.

Quantization

After fine-tuning, I applied quantization to optimize the model for deployment in resource-constrained environments and improve speed. Quantization reduces the model’s memory footprint by representing its weights with fewer bits, making it faster and more efficient while maintaining acceptable performance. Using the llama.cpp library, I experimented with three popular quantization levels to balance model size and accuracy: 2-bit, 4-bit, and 8-bit quantization [166, 167, 168]. Each level presents different trade-offs between memory usage and performance, with 2-bit offering the highest compression and 8-bit maintaining nearly full accuracy.

These quantized versions let me assess the trade-off between effectiveness (accuracy) and efficiency (model size and inference speed) for various deployment scenarios in REST API testing. The 2-bit quantization is ideal for highly memory-constrained devices such as edge environments (e.g., 2.96 GB), 4-bit strikes a balance for mid-tier resources (e.g., 4.58 GB), and 8-bit is suited for relatively high-resource environments where performance is a priority (e.g., 7.95 GB).

7.2.4 REST API Testing

Figure 7.1 illustrates the LlamaRestTest framework, which integrates the ARAT-RL approach for adaptive REST API testing. A key component of ARAT-RL is the Q-Table,

which prioritizes API operations and parameters. The *Q-Table Updater* updates Q-values based on responses received from API requests. Specifically, successful responses (2xx) reduce the Q-value, encouraging exploration of other operations, while failure responses (4xx or 500) increase the Q-value, directing further testing toward problematic areas. This update follows the Q-learning formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (7.1)$$

where α is the learning rate, γ is the discount factor, and r is the reward assigned based on the response. As shown in Figure 7.1, the *Request Sender* interacts with the *Q-Table Updater*, dynamically adjusting priorities to optimize testing coverage.

Once initialized, ARAT-RL applies an ϵ -greedy strategy to balance exploration (testing less-explored operations) and exploitation (focusing on previously promising combinations). The *Operation Selector* chooses an API operation based on its Q-value, followed by the *Parameter Selector*, which assigns probabilities to parameters, with more frequently used parameters being favored. The *Value Generator* produces parameter values, typically chosen randomly but in conformance with the expected types.

When repeated failures occur (e.g., multiple 4xx responses), LlamaREST-IPD and LlamaREST-EX are activated. The LlamaREST-IPD model is prompted in the same format employed during fine-tuning:

```
<s>[INST] Find Inter-Parameter Dependency for the parameter below
name: {parameter name}
description: {parameter description and server response message} [/INST]
```

This prompt directs the model to analyze parameter descriptions and server response messages, identifying dependencies, such as when a parameter is required only if another parameter is present.

Similarly, LlamaREST-EX generates example values for parameters using the following prompt structure, which was used for fine-tuning:

```
<s>[INST] Find example values for the parameter below in a list format
name: {parameter name}
description: {parameter description and server response message} [/INST]
```

This enables LlamaREST-EX to produce valid, realistic parameter values based on the available descriptions and server responses.

After parameters and values are refined, the *Request Sender* submits the API request. Depending on the API response, the *Q-Table Updater* adjusts the priorities of the operation-parameter combinations. Successful requests lower the priority of that combination, encouraging exploration, whereas failures assign positive rewards, driving further probing of problematic parts of the API.

7.3 Evaluation

In this section, I present the results of empirical studies conducted to assess LlamaRestTest, focusing on answering the following research questions:

1. **RQ1:** How do LlamaREST-EX and LlamaREST-IPD compare with the state-of-the-art REST API testing assistant tool, RESTGPT, in terms of identifying inter-parameter dependencies and generating valid inputs for REST API testing?
2. **RQ2:** How do fine-tuning and quantization affect REST API testing in terms of achieved code coverage and operation coverage?
3. **RQ3:** How does LlamaRestTest compare with state-of-the-art REST API testing tools in terms of achieved code coverage and operation coverage?
4. **RQ4:** In terms of error detection, how does LlamaRestTest perform in triggering 500 (Internal Server Error) responses compared to state-of-the-art REST API testing tools?

Table 7.1: REST services used in the evaluation.

REST Service	Lines of Code	Operations
Features-Service	1688	18
Genome-Nexus	22143	23
Language-Tool	113170	2
Market	7945	13
Person-Controller	601	12
Project-Tracking-System	3784	67
Rest-Countries	1619	22
User-Management	2805	22
YouTube	2422	1
FDIC	–	6
Ohsome	–	122
Spotify	–	12

5. **RQ5:** How do analysis of parameter description, analysis of server response messages, LlamaREST-IPD, and LlamaREST-EX contribute to the overall performance of LlamaRestTest in terms of code coverage and operation coverage?

7.3.1 Experiment Setup

The experiments were conducted on an M1 MacBook Pro running Sonoma 14.4.1 with 64GB of memory. I installed all the services from the benchmark suite, along with the REST API testing tools I intended to evaluate. To prevent any dependency conflicts, I restored the database for each service before running each test. Each service and tool was hosted with default settings and tested sequentially to avoid interference. I continuously monitored CPU and memory usage to ensure that resource limitations did not impact the performance of the testing tools. Each tool was run for one hour, a standard time budget used in the area [19, 20, 15, 17, 163], to provide a fair comparison. To minimize the effect of randomness on the results, I repeated each test 10 times and averaged the outcomes.²

For the evaluation, I relied on the same set of REST API testing tools and services used by ARAT-RL [15]. Accordingly, I compared AutoRestTest with ARAT-RL [15], EvoMaster [45], MoRest [17], and RESTler [9]. Specifically, I used the latest released version

²For the ablation study (RQ5), I performed only three runs due to its extensive evaluation time.

or the latest commit when a release was unavailable: RESTler v9.2.4, EvoMaster v3.0.0, ARAT-RL v0.1, MoRest (obtained directly from the authors).

The ARAT-RL benchmark dataset consists of 10 RESTful services. In addition to these services, I included services from the RESTGPT study [160] due to the availability of a ground truth of extracted rules. Out of the total 16 services, I excluded SCS and NCS because they were written by EvoMaster’s authors, and I aimed to avoid potential bias. I also excluded OCVN due to authentication issues. Lastly, I excluded OMDB, which is a toy online service with only one API operation that all testing tools can process in a second. Ultimately, I used 12 services: Features Service, Language Tool, REST Countries, Genome Nexus, Person Controller, User Management Microservice, Market Service, Project Tracking System, Ohsome,³ YouTube-Mock, and Spotify. Table 7.1 provides details on these services, including the lines of code (for the open-source services) and the number of API operations.

The key metrics for evaluating effectiveness and fault-detection ability included code coverage, the number of successfully processed operations in the specification, and the number of 500 status codes encountered, which are among the most commonly used metrics [163]. To measure code coverage, I used JaCoCo [114]. The number of processed operations and the number of errors detected (500 status codes) were collected using a script from the ARAT-RL repository [150]. That script tracks 500 status codes for each operation and removes duplicate 500 errors based on server response messages.

7.3.2 RQ1: Comparison with RESTGPT

To assess the effectiveness of LlamaREST-IPD in identifying IPD rules and LlamaREST-EX in generating semantically valid testing inputs, I compared both models with RESTGPT, a state-of-the-art tool for enhancing specifications. I also evaluated Llama3-8B, the base model for both LlamaREST-IPD and LlamaREST-EX, along with various quantiza-

³Ohsome has an open-source version [173], but I used the online version [174] as the open-source version consistently produced server errors.

Table 7.2: Comparison of semantically valid parameter value generation by LlamaREST-EX, Llama3-8B, and RESTGPT (‘FT’ indicates fine-tuning and ‘Q’ indicates quantization).

API	LlamaREST-EX				RESTGPT	Llama3-8B
	FT with Q(2-Bit)	FT with Q(4-Bit)	FT with (Q 8-Bit)	FT		
Genome-Nexus	10.66%	35.29%	35.48%	57.02%	36.25%	9.00%
Language-Tool	22.58%	58.82%	64.52%	69.23%	82.98%	60.00%
REST-Countries	33.33%	78.79%	75.47%	79.22%	75.29%	46.84%
YouTube	14.75%	67.27%	70.31%	73.08%	73.33%	21.67%
FDIC	21.21%	33.33%	47.46%	54.69%	70.42%	14.29%
Ohsome	27.59%	74.14%	93.75%	92.16%	76.83%	2.78%
Spotify	73.74%	73.81%	75.86%	81.71%	66.67%	0.00%
Average	29.12%	60.21%	66.12%	72.44%	68.82%	22.94%

Table 7.3: Comparison of inter-parameter dependency rule generation performance among LlamaREST-IPD, RESTGPT, and Llama3-8B (‘FT’ indicates fine-tuning, ‘Q’ indicates quantization, ‘O’ indicates correct rule generation, and ‘-’ indicates failure to generate the rule).

Service Name	Parameter Name	Rule	LlamaREST-IPD				RESTGPT	Llama3-8B
			FT with Q(2-Bit)	FT with Q(4-Bit)	FT with Q(8-Bit)	FT		
Language-Tool	data	Or(text,data)	-	-	-	-	O	-
	preferredVariants	IF preferredVariants THEN language==auto	-	-	-	-	-	-
YouTube	eventType	IF eventType THEN type=video	-	O	O	O	O	O
	forMine	IF forMine==true THEN type=video	-	-	-	O	O	-
	videoLicense	IF videoLicense THEN type=video	-	-	-	O	-	-
	videoType	IF videoType THEN type=video	-	O	O	O	O	O
	videoEmbeddable	IF videoEmbeddable THEN type=video	O	-	O	O	O	-
	videoDimension	IF videoDimension THEN type=video	-	O	O	O	O	-
	location	AllOrNone(this.locationRadius)	-	-	-	-	-	-
	videoDefinition	IF videoDefinition THEN type=video	-	-	-	O	-	-
	relatedToVideoId	IF relatedToVideoId THEN type=video	-	-	-	-	-	-
	videoCaption	IF videoCaption THEN type=video	-	-	O	O	O	-
	forContentOwner	IF forContentOwner THEN onBehalfOfContentOwner	-	-	-	-	-	-
	videoDuration	IF videoDuration THEN type=video	-	O	O	O	O	-
videoSyndicated	IF videoSyndicated THEN type=video	-	O	O	O	-	-	
videoCategoryId	IF videoCategoryId THEN type=video	-	O	-	O	-	-	
Spotify	collaborative	IF collaborative==true THEN public==false	-	-	-	O	O	-
Total		18	1	6	7	12	9	2

tion variants (2-bit, 4-bit, and 8-bit). To measure effectiveness, I used as ground truth, the RESTGPT dataset of extracted rules, including IPDs and example values [160]. The RESTGPT dataset contains manually-extracted rules from OpenAPI specifications of nine services, but IPD and value constraint rules occur for only three and seven of the services, respectively; for the other services, these rules were not described in their OpenAPI specifications. Thus, the evaluation of value generation is based on seven of the services from the RESTGPT benchmark, whereas the evaluation of IPD rule extraction is based on three of the services.

Table 7.2 presents the results on precision of semantically valid value generation for LlamaREST-EX, RESTGPT, and Llama3-8B for seven services. Table 7.3 further narrows the focus to three of these services, presenting the available IPD rules identified through

parameter descriptions.

I evaluated value generation based on semantic validity, as proposed by a recent study and adopted by RESTGPT: a semantically valid input must be coherent with the API domain [50]. For instance, “Berlin” is a valid input for the parameter `addressLocality` in DHL’s API, whereas “dog” is not. For RESTGPT, I used the generated values from its artifact; for Llama3-8B, I applied the same prompts used by RESTGPT. Precision was determined by two external developers independently assessing whether the generated values were semantically valid, with any discrepancies resolved through discussion to reach a consensus.

Llama3-8B, although competitive on some APIs, generally performs poorly, with an average accuracy of just 24.94%. It struggles significantly on APIs such as Spotify (0%) and Ohsome (2.78%), underscoring its limitations in generating semantically valid values. However, fine-tuning the model, as demonstrated with LlamaREST-EX, leads to significant performance improvement.

LlamaREST-EX, especially its FT version, shows substantial gains over Llama3-8B, achieving an average accuracy of 72.44%. For example, on the Ohsome API, the FT version of LlamaREST-EX reaches 92.16%, outperforming RESTGPT’s 76.83%; similarly, on Genome-Nexus, it scores 57.02%, achieving considerably higher precision than RESTGPT (36.25%). Overall, LlamaREST-EX with fine-tuning outperforms RESTGPT, which achieved 68.82% accuracy, compared to LlamaREST-EX’s 72.44%. Even the quantized versions of LlamaREST-EX (4-bit and 8-bit) show reasonable performance, with 60.21% and 66.12% precision, respectively.

LlamaREST-EX generally performs well, but there are two cases where it is less effective than RESTGPT: Language-Tool and FDIC. These results highlight some limitations of the approach and provide insights for future improvements. For Language-Tool, the low performance is primarily due to LlamaREST-EX’s difficulty in handling complex parameter constraints, particularly for region-specific formats. For example, with the

`altLanguage` parameter, LlamaREST-EX output “en” instead of the correct “en-GB”. RESTGPT, leveraging its larger model size, was able to correctly interpret the format specification from the parameter description. In the case of FDIC, the lower performance came from the specialized nature of the API, which requires more domain-specific financial knowledge that the fine-tuning process did not fully capture.

These cases highlight the trade-off between model size and performance. Although the approach with LlamaRestTest demonstrates that fine-tuning a smaller, open-source model can significantly enhance task-specific effectiveness and make it competitive with much larger models in many cases, there are still scenarios where larger models can have an advantage in handling very specific or complex constraints.

Table 7.3 presents the IPD rules identified through parameter descriptions for three services among the seven services in Table 7.2. Llama3-8B demonstrates limited capability in detecting IPD rules, identifying only two rules across all services. LlamaREST-IPD, particularly in its fine-tuned version, significantly outperforms both Llama3-8B and RESTGPT. The FT variant of LlamaREST-IPD identifies 12 correct IPD rules, compared to nine by RESTGPT, highlighting its ability to capture more complex relationships. Even the quantized versions of LlamaREST-IPD (4-bit and 8-bit) demonstrate consistent improvements in performance, but the 2-bit version has considerable performance degradation, identifying only one IPD rule correctly.

Fine-tuning significantly improves the accuracy of semantically valid input generation and IPD rule detection, outperforming both the base Llama3-8b model and the state-of-the-art RESTGPT tool. Moreover, with 8-bit and 4-bit quantization, the fine-tuned model remains competitive with RESTGPT, while still performing considerably better than the base model.

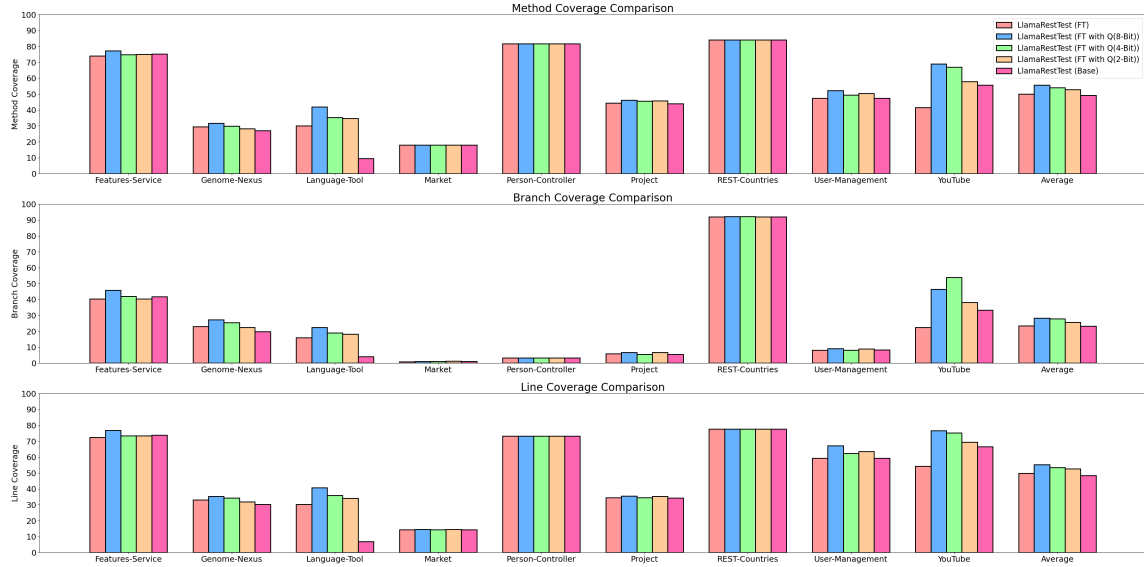


Figure 7.2: Code coverage achieved on 10 runs for open-source services using LlamaRestTest configured with the base Llama3-8B model, the fine-tuned model (FT), and the quantized models (Q).

Table 7.4: Average number of operations processed in 10 runs for closed-source services using LlamaRestTest configured with Llama3-8B (Base), fine-tuning (FT), and quantization (Q).

Service	FT with 8-Bit Q	FT with 4-Bit Q	FT with 2-Bit Q	FT	Base
FDIC	6.0	6.0	6.0	6.0	6.0
Ohsome	24.5	21.4	13.2	12.1	11.2
Spotify	7.0	7.0	6.2	6.4	5.8
Total	37.5	34.4	25.4	24.5	23.0

7.3.3 RQ2: Impact of Fine-Tuning and Quantization on REST API Testing

To evaluate the impact of fine-tuning and various quantization levels on REST API testing, I assessed LlamaRestTest under five configurations: the base Llama3-8B model, the fine-tuned version (at full precision), and the quantized versions at 8-bit, 4-bit, and 2-bit quantization. I evaluated the effectiveness of these configurations in terms of code coverage achieved for the open-source services and operations processed for the closed-source services in the benchmark (Table 7.1).

Figure 7.2 presents a comparison of method, branch, and line coverage across multiple services for the five LlamaRestTest configurations: the fine-tuned version, the fine-tuned

Table 7.5: Average number of operations processed for closed-source services by different testing tools (LlamaRestTest configured with 8-bit quantization.)

Service	MoRest	RESTler	ARAT-RL	EvoMaster	LlamaRestTest
FDIC	6.0	6.0	6.0	6.0	6.0
Ohsome	0.0	0.0	0.0	0.0	24.5
Spotify	3.9	4.6	5.6	5.2	7.0
Total	9.9	10.6	11.6	11.2	37.5

versions with 8-bit, 4-bit, and 2-bit quantization, and the base model. Each panel represents one of the coverage metrics—method, branch, or line—for the nine open-source services in the benchmark. The figure also shows the average coverage, across services, for each configuration. The fine-tuned and quantized versions generally exhibited higher coverage compared to the base model, with significant gains observed for the Language-Tool and YouTube services in particular.

The results show that fine-tuning alone produced modest improvements over the base model. The fine-tuned model achieved average coverage rates of 50.11% for method coverage (compared to 49.21% for base), 23.55% for branch coverage (compared to 23.24% for base), and 49.91% for line coverage (compared to 48.51% for base). This represents improvements of approximately 0.9%, 0.3%, and 1.4% respectively, demonstrating that fine-tuning provides an observable but limited enhancement to testing capability.

On combining fine-tuning with quantization, I observed substantially greater improvements. LlamaRestTest with 8-bit quantization exhibited 6.6% improvement in method coverage compared to the base model, reaching 55.8% coverage on average. The 4-bit quantized version resulted in coverage gain of 4.8% (54.0% method coverage), while the 2-bit quantized version achieved 3.7% increase (52.9% method coverage). The trends are similar for branch and line coverage, with the quantized versions performing better than the fine-tuned model in gains over the base model. In terms of branch coverage, the 8-bit, 4-bit, and 2-bit quantized versions showed increases of 5.1%, 4.7%, and 2.5%, respectively, over the base model; likewise, for line coverage, the gains of these quantized version were 6.8%,

5.0%, and 4.1%, respectively.

Table 7.4 shows the number of successfully processed operations for the closed-source services in the benchmark under the five LlamaRestTest configurations. LlamaRestTest with fine-tuning alone processed 24.5 operations, which is 1.5 operations more than the base model's 23 operations. More significantly, LlamaRestTest with 8-bit quantization processed 37.5 operations (14.5 operations more than the base model's 23), while the 4-bit quantized version processed 34.4 operations (11.4 operations more than the base model). Finally, the 2-bit version was also more effective than the base model, processing 25.4 operations (2.4 operations more than the base model).

These results demonstrate that fine-tuning alone provides modest benefits, but the combination of fine-tuning with appropriate quantization delivers significantly better performance. There are two main contributing factors for this phenomenon. First, the faster inference times of quantized models enable more test generation within the same testing budget, processing for instance 22–81% more inference tasks in the same time budget [175]. For example, for IPD rule extraction (RQ1), the fine-tuned model required 48.9 seconds per IPD on average, whereas the 8-bit quantized version reduced the inference time to 36.9 seconds. The 4-bit and 2-bit quantized models further reduced the time to 26.2 and 26.1 seconds, respectively. The smaller models benefit from faster inference, which enhances the effectiveness of REST API testing via more efficient processing of server response messages. Second, quantization introduces a regularization effect that may help the model generalize better with various answers rather than memorizing specific examples [176].

Although quantized models generally performed well, I observed scenarios where the 2-bit quantized model underperformed compared to the non-quantized model. Specifically, for services, such as Genome-Nexus and Spotify, that rely heavily on domain-specific knowledge, the 2-bit quantized model often generated incorrect outputs, leading to lower code coverage. This suggests that the extreme compression in the 2-bit model may result in a loss of precision that affects performance in complex, domain-specific scenarios. How-

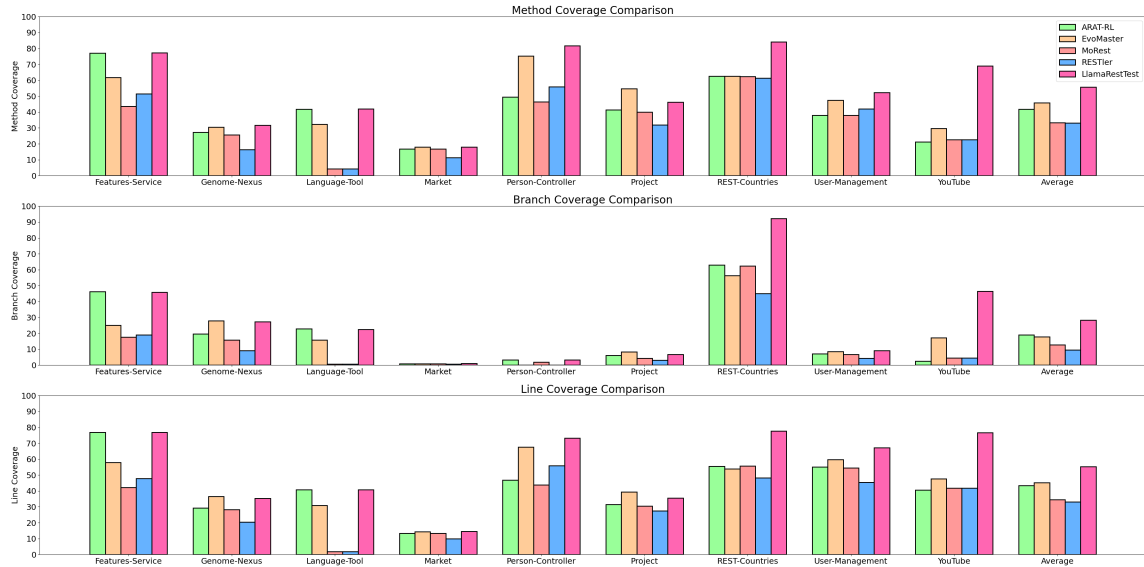


Figure 7.3: Code coverage achieved on 10 runs for open-source services with MoRest, RESTler, ARAT-RL, EvoMaster, and LlamaRestTest (fine-tuned with 8-bit quantization).

ever, the 4-bit and 8-bit quantized models consistently outperformed the non-quantized models, even in these challenging cases. This indicates that these quantization levels strike a good balance between model compression and maintaining the necessary precision for effective API testing across various domains.

LlamaRestTest with fine-tuning and 8-bit quantization achieved the highest code and operation coverage, processing 37.5 operations while achieving improvements of 5.5%, 5.1%, and 6.8% in method, branch, and line coverage, respectively, over the base model, thus showing a good balance between efficiency and effectiveness. Fine-tuning alone provided modest gains over the base model. While 4-bit and 8-bit quantized models consistently outperformed non-quantized versions, the 2-bit model showed limitations in complex, domain-specific scenarios, highlighting the trade-off between model compression and precision for REST API testing.

7.3.4 RQ3: Comparison with State-of-the-art REST API Testing Tools

Next, I compared LlamaRestTest against four state-of-the-art REST API testing tools in terms of code coverage (for the open-source services) and operation coverage (for the closed-source services). For this study, the testing tools used enhanced versions of OpenAPI specifications of the services generated by RESTGPT [160].

Figure 7.3 presents the code coverage achieved by MoRest, RESTler, ARAT-RL, EvoMaster, and LlamaRestTest on the open-source services in the benchmark. The results show that LlamaRestTest consistently outperforms all other tools in method, branch, and line coverage. For method coverage, LlamaRestTest achieved 55.8%, exceeding MoRest (33.3%) by 22.5 percentage points, RESTler (33.1%) by 22.7 percentage points, ARAT-RL (41.7%) by 14.1 percentage points, and EvoMaster (45.8%) by 10 percentage points. In terms of branches, LlamaRestTest reached 28.3% coverage, achieving 15.6 increase over MoRest, 18.7 increase over RESTler, 9.2 increase over ARAT-RL, and 10.5 increase over EvoMaster. Finally, for line coverage, LlamaRestTest achieved 55.3% coverage, again outperforming MoRest (+20.7), RESTler (+22.1), and ARAT-RL (+11.9), and EvoMaster (+10) considerably.

Table 7.5 presents results on operation coverage, showing the average number of operations processed by each tool for the closed-source services. Like the results on code coverage, LlamaRestTest outperformed the other tools by a wide margin, successfully processing 37.5 operations overall. This represents improvements of 27.6 operations over MoRest (9.9), 26.9 operations over RESTler (10.6), 25.9 operations over ARAT-RL (11.6), and 26.3 operations over EvoMaster (11.2).

LlamaRestTest's effective performance comes from its ability to analyze server responses, which allows it to detect errors that other tools might miss. For example, in testing APIs like Ohsome and Spotify, LlamaRestTest was able to process more operations by utilizing information from server messages. In the case of Ohsome, where the server returned a message indicating that users must specify only one of three parameters, LlamaRestTest

used this information to test the operations that the other tools missed. Conversely, LlamaRestTest may be less effective on services that provide minimal or poor server feedback. In such cases, LlamaRestTest could spend time analyzing response messages that do not contain useful information for refining requests, potentially reducing its efficiency.

These results demonstrate that leveraging server response feedback with LlamaREST-IPD and LlamaREST-EX significantly improves testing effectiveness compared to current state-of-the-art tools, even when using enhanced specifications generated by RESTGPT. The ability to interpret and utilize server responses gives LlamaRestTest an edge in most scenarios, particularly those with informative server feedback.

LlamaRestTest significantly outperforms other API testing tools, achieving 10.0%–22.7% more method coverage, 9.2%–18.7% more branch coverage, 10.0%–22.1% more line coverage, and processing 2.5x–3x more operations than competing tools. It performs particularly well in scenarios with informative server feedback, but may be less effective, perform similarly to other tools, when testing APIs with minimal response messages or messages that lack useful information.

7.3.5 RQ4: Fault-Detection Capability

Table 7.6 presents the results on fault-detection capability of five REST API testing tools—MoRest, RESTler, ARAT-RL, EvoMaster, and LlamaRestTest—measured as the number of 500 internal server errors detected across all services in the benchmark. To measure unique server errors, I used the error-counting script provided in the ARAT-RL artifact [150], which automatically counts the 500 errors in the server log and discards duplicate entries.

LlamaRestTest demonstrated the highest fault-detection ability, discovering 204 faults in 10 runs on the services in the benchmark. This represents 64 more faults detected than MoRest (140), 74 more than RESTler (130), 44 more than ARAT-RL (160), and 74 more

Table 7.6: Internal server errors detected over 10 runs.

Service	MoRest	RESTler	ARAT-RL	EvoMaster	LlamaRestTest
Features-Service	10	10	10	10	10
Genome-Nexus	10	0	10	0	10
Language-Tool	0	0	10	10	10
Market	10	10	10	10	10
Person-Controller	80	80	80	80	80
Project-Tracking-System	10	10	10	10	10
REST-Countries	10	10	10	0	10
User-Management	10	10	10	10	10
YouTube	0	0	0	0	0
FDIC	0	0	10	0	10
Ohsome	0	0	0	0	36
Spotify	0	0	0	0	8
Total	140	130	160	130	204

than EvoMaster (130). The results clearly show that LlamaRestTest outperforms all other tools, detecting more faults across diverse services.

The most notable detected errors were in the Ohsome and Spotify services. For Ohsome, LlamaRestTest was the only tool to detect faults, identifying 36 errors where all other tools failed. Similarly, LlamaRestTest was the only tool to find faults in Spotify, detecting 8 errors where all others found none. Both if these instances involve processing of complex scenarios. For example, the `GET /playlists/playlist_id/tracks` operation in Spotify’s API requires specific knowledge of how Spotify generates `playlist_id`. Spotify generates IDs for playlists that are typically 22 characters long, with constraints on the permitted characters and patterns. While many tools fail to generate valid IDs, LlamaREST-EX, due to its fine-tuning, accurately retrieved and generated valid Spotify playlist IDs. As a result, LlamaRestTest was able to successfully execute the `GET /playlists/playlist_id/tracks` operation, triggering a chain of interactions across the service. For instance, after retrieving the International Standard Recording Code (ISRC) from the playlist’s tracks, the mutation process randomly selects an ISRC to use as `user_id` in the subsequent `GET /users/user_id/playlists` operation. This sequence exposes a hidden dependency conflict, which causes an Internal Server Error.

Table 7.7: Ablation study on the impact of key components of LlamaRestTest on coverage achieved.

	Method	Branch	Line
LlamaRestTest	55.8%	28.3%	55.3%
1. Without Parameter Description	49.9% (-5.9%)	26.9% (-1.4%)	50.8% (-4.5%)
2. Without Server Response	49.7% (-6.1%)	23.9% (-4.4%)	49.0% (-6.3%)
3. Without LlamaREST-IPD	46.6% (-9.2%)	22.1% (-6.2%)	46.9% (-8.4%)
4. Without LlamaREST-EX	52.1% (-3.7%)	23.8% (-4.5%)	52.3% (-3.0%)

LlamaRestTest outperforms state-of-the-art REST API testing tools in fault detection by detecting 204 Internal Server Error (500) responses over 10 runs, thus identifying 44 to 74 more errors than the other tools, which illustrates its superior ability to detect internal server errors.

7.3.6 RQ5: Ablation Study

To assess the contribution of the key components of LlamaRestTest, I conducted an ablation study by systematically removing specific features and measuring the resulting effect on method, branch, and line coverage across three runs. In particular, I disabled the following features individually to study their effects on coverage: analysis of parameter descriptions, analysis of server response messages, LlamaREST-IPD, and LlamaREST-EX. Table 7.7 presents the code coverage achieved by LlamaRestTest with and without these components, thus showing how much each feature contributes to its overall performance.

The full LlamaRestTest system achieved method coverage of 55.8%, branch coverage of 28.3%, and line coverage of 55.3%. Removing the parameter description feature resulted in a noticeable drop, with method coverage decreasing by 5.9%, branch coverage by 1.4%, and line coverage by 4.5%. This suggests that including parameter descriptions improves test case generation by providing relevant information for generation of valid inputs. Interestingly, excluding the server response feature caused an even more significant drop, particularly in branch coverage, which decreased by 4.4%; method and line coverage fell by 6.1% and 6.3%, respectively. While previous studies in this area have focused primarily

on enhancing specifications [144, 160], these results highlight the critical role of server feedback in guiding test case generation and increasing coverage via analysis of dynamic behaviors at runtime.

The removal of LlamaREST-IPD had the most substantial impact on performance, reducing method coverage by 9.2%, branch coverage by 6.2%, and line coverage by 8.4%. This suggests that modeling inter-parameter dependencies is crucial for generating more effective test cases that cover complex interactions between parameters. Finally, excluding LlamaREST-EX resulted in moderate reductions in coverage, with method, branch, and line coverage decreasing by 3.7%, 4.5%, and 3.0%, respectively. But LlamaREST-EX still contributes meaningfully to overall test effectiveness, particularly in terms of branch coverage.

The ablation on key components of LlamaRestTest shows that, while each component contributes to the overall performance of LlamaRestTest, server response analysis and inter-parameter dependency modeling are the more critical components, with their removal causing larger drops in code coverage than the other components.

7.3.7 Threats to Validity

Like any empirical study, there are potential threats to the validity of the results. The selection of REST APIs may not be representative and affect external validity. Although I used a diverse set of 12 real-world services, including services that have been used in prior evaluations of REST API testing tools, LlamaRestTest may perform differently on other benchmarks containing services with different characteristics. The reliance on specific fine-tuning datasets, such as Martin-Lopez’s IPD database and APIs-guru, may affect the performance of the fine-tuned models on other services. In terms of tool selection, I used the state-of-the-art REST API testing tools available at the time of this study. I note, however, that concurrently to the work, two new REST API testing techniques have been

proposed [177, 48]. I could not compare with either of these approaches because they had not yet been published when I performed this work.

There are potential issues related to the experiment setup that could affect the accuracy of the results. The settings used for the testing tools and the services might not be optimal. I attempted to minimize these effects on the results by using the same settings that were used in a previous study [15]. I hosted each testing tool and service on separate machines to prevent interference. To account for randomness in tool results, I ran each tool 10 times on each service (this was done for all studies except the ablation study). The use of automated scripts to measure fault detection and code coverage could introduce inaccuracies due to bugs in the scripts. However, such factors are unlikely to affect the results in any significant way as I used scripts that have been used in prior studies [150]. Moreover, I manually checked some of the testing results to validate them. The manual validation step of inter-parameter dependency rules for creating the fine-tuning dataset, while improving accuracy, introduces potential inconsistencies and limits scalability.

LlamaRestTest’s integration with ARAT-RL creates tool dependency where potential issues in ARAT-RL could affect LlamaRestTest’s performance. Again, I thoroughly tested and spot checked the implementation. I also make LlamaRestTest and the experiment dataset available in the artifact [169], which allows other researchers to validate and extend the evaluation.

7.4 Conclusion

I introduced LlamaRestTest, a new REST API testing approach that leverages the capabilities of large language models through fine-tuning and quantization to improve testing effectiveness and efficiency. LlamaRestTest incorporates two key components, LlamaREST-IPD and LlamaREST-EX, which focus on inter-parameter dependency detection and realistic parameter value generation. The evaluation, performed on 12 real-world REST services, shows that LlamaRestTest can outperform state-of-the-art tools such as RESTler, EvoMas-

ter, MoRest, and ARAT-RL, achieving significantly higher code coverage and processing more service operations than those tools. Additionally, the ablation study highlights the importance of each component, with LlamaREST-IPD and server response messages being critical to achieving the best results. The results also show that fine-tuning significantly enhances LlamaRestTest's performance, even outperforming larger models such as REST-GPT which is based on ChatGPT-3.5 Turbo. Quantization further improves the model's efficiency without substantial loss in accuracy, making LlamaRestTest both powerful and practical for real-world use.

CHAPTER 8

A MULTI-AGENT APPROACH FOR REST API TESTING WITH SEMANTIC GRAPHS AND LLM-DRIVEN INPUTS

8.1 Introduction

Existing tools often achieve low coverage, especially on large REST services, as shown in recent studies (e.g., Language Tool, Genome Nexus, and Market in the ARAT-RL evaluation [15], and Spotify and OhSome in the NLP2REST evaluation [144]). To overcome these limitations, I propose AutoRestTest, a new approach that integrates a semantic property dependency graph (SPDG) and multi-agent reinforcement learning (MARL) with Large Language Models (LLMs) to enhance REST API testing. The "agent" refers to a specialized software component that perceives its environment through sensors, makes decisions based on its knowledge and objectives, and acts upon the environment to achieve specific goals [178, 179]. Unlike fully autonomous agents in distributed systems, the agents in AutoRestTest are cooperative components of a centralized system that collaborate to solve different aspects of the REST API testing problem while sharing a common overall objective. This definition aligns with the use of agents in multi-agent reinforcement learning frameworks for complex problem-solving [43, 180], where problem decomposition and specialized learning improve overall system performance.

Specifically, AutoRestTest employs four specialized agents to optimize the testing process. The *dependency agent* manages and utilizes the dependencies between operations identified in the SPDG, guiding the selection of dependent operations for API requests. The *operation agent* selects the next API operation to test, prioritizing operations likely to yield valuable test results based on previous results, such as successfully processed dependent operations. The *parameter agent* chooses parameter combinations for the selected

operation to explore different configurations. Finally, the *value agent* manages the generation of parameter values using three data sources: values from dependent operations, LLM-generated values that satisfy specific constraints using few-shot prompting [25], and type-based random values. The value agent learns which data source is most effective for each parameter type and context.

The AutoRestTest agents collaborate to optimize the testing process using the multi-agent value decomposition Q-learning approach [180, 181]. When selecting actions, each agent is employed independently using the epsilon-greedy strategy for exploitation-exploration balancing. However, during policy updates using the Q-learning equation, AutoRestTest uses value decomposition to consider the joint actions across all agents. Through centralized policy updates, each agent converges toward selecting the optimal action while accounting for the actions of the other agents.

I evaluated AutoRestTest on 12 real-world RESTful services used in previous studies [144, 15], including well-known services such as Spotify, and compared its performance with that of four state-of-the-art REST testing tools, recognized as top-performing tools in recent studies [19, 15, 17, 20]: RESTler [9], EvoMaster [45], MoRest [17], and ARAT-RL [15]. To ensure a fair comparison, I used enhanced API specifications generated by RESTGPT [160], which augments REST API documents with realistic input values generated using LLMs. To measure effectiveness, I used code coverage for the open-source services, operation coverage for the online services, and fault detection ability—measured as internal server errors triggered—for all the services. These are the most commonly used metrics in this field [163, 19].

AutoRestTest demonstrated superior performance across all coverage metrics compared to existing tools. It achieved 58.3% method coverage, 32.1% branch coverage, and 58.3% line coverage, significantly outperforming ARAT-RL, EvoMaster, RESTler, and MoRest by margins of 12–27%. For closed-source services, AutoRestTest processed 25 API operations, compared to 9–11 operations processed by the other tools. These results show that

AutoRestTest can perform more comprehensive API testing than the other tools.

In terms of fault detection, AutoRestTest identified 42 operations with internal server errors, outperforming ARAT-RL (33), EvoMaster (20), MoRest (20), and RESTler (14). Notably, AutoRestTest was the only tool that detected an internal server error for Spotify [182]. I reported the errors detected on FDIC, OhSome, and Spotify, as these are actively maintained projects. The OhSome error has been confirmed and fixed [183]; I am still waiting for feedback from the developers for the other bug reports.

I also performed an ablation study, which revealed the importance of AutoRestTest’s key components. Removing temporal-difference Q-learning caused the largest performance drop, with method, line, and branch coverage falling to 45.6% (-12.7), 18.2% (-13.9), and 45.8% (-12.5), respectively. SPDG removal reduced coverage to 46.7% (-11.6), 18.7% (-13.4), and 47.6% (-10.7), while LLM removal led to 47.4% (-10.9), 19.3% (-12.8), and 45.8% (-12.5). Overall, removing any component decreased coverage by 10.7–13.9%, demonstrating the significance of each component in contributing to AutoRestTest’s effectiveness.

The main contributions of this work are:

- A novel REST API testing technique that reduces the operation dependency search space using a similarity-based graph model, employs multi-agent reinforcement learning to consider optimization among the testing steps, and leverages LLMs to generate realistic test inputs.
- Empirical results showing that AutoRestTest outperforms state-of-the-art REST API testing tools—even when provided with enhanced API specifications—by covering more operations, achieving higher code coverage, and triggering more failures.
- An artifact including the AutoRestTest tool, the benchmark services used in the evaluation, and detailed empirical results, which serves as a comprehensive resource for supporting further research and replication of the results [184].

8.2 Motivating Example

Figure 8.1 depicts the `/register` endpoint in the Market API’s OpenAPI specification. This endpoint is vital for creating user credentials needed in other components of the API. Existing REST API testing tools struggle to generate valid requests for this operation due to the strict parameter requirements: `email`, `name`, and `password` are required parameters, each with specific constraints on valid values, while `links` is an optional parameter. Moreover, these tools fail to prioritize information gained from a successful operation invocation in subsequent requests.

AutoRestTest addresses these issues through a multi-agent approach. The parameter agent employs reinforcement learning to identify valid parameter combinations, learning that `(email, name, password)` is a required parameter set while avoiding invalid combinations that would trigger 400 errors. For these parameters, the value agent first determines the appropriate data source for each parameter—choosing between LLMs, dependency values from previous operations, or random values—because different parameters require different generation strategies to create valid values. For the `/register` endpoint, it selects LLM generation as the parameters require specific formats (e.g., email format, password rules) that basic defaults cannot match, and dependency values are not available for this initial operation. It then generates context-aware values that comply with the specification’s validation patterns. After a successful registration, the dependency agent utilizes the SPDG to identify operations that depend on user credentials (e.g., `/customer/cart`, `/customer/orders`) and propagates the registered user’s information to these dependent operations. The operation agent then prioritizes testing these dependent operations, while the parameter and value agents reuse the strategies that were successful for the `/register` endpoint. Through value decomposition, MARL enables efficient policy updates by appropriately distributing rewards to each agent based on their contribution.

```

1 /register:
2   post:
3     tags:
4       - customer-rest-controller
5     summary: createCustomer
6     operationId: createCustomerUsingPOST
7     requestBody:
8       description: user
9       content:
10        application/json:
11          schema:
12            type: object
13            properties:
14              links:
15                type: array
16                items:
17                  $ref: '#/components/schemas/Link'
18            email:
19              type: string
20              maxLength: 50
21              pattern: "^[\\w-]+(\\.([\\w-]+))*@([\\w-]+\\.)+[a-zA-Z]+$"
22            name:
23              type: string
24              maxLength: 50
25              pattern: "^[\\pL ' -]+$"
26            password:
27              type: string
28              maxLength: 50
29              minLength: 6
30              pattern: "^[a-zA-Z0-9]+$"
31     responses:
32       "201":
33         description: Created
34         content:
35           '*/*':
36             schema:
37               $ref: '#/components/schemas/UserDTORes'
38       "401":
39         description: Unauthorized
40       "403":
41         description: Forbidden
42       "404":
43         description: Not Found

```

Figure 8.1: A part of Market API's OpenAPI Specification.

8.3 Approach

8.3.1 Overview

Figure 8.2 illustrates the architecture of AutoRestTest, highlighting its core components: the SPDG, the REST agents, and the request generator. The overall workflow consists mainly of two phases: initialization and testing execution.

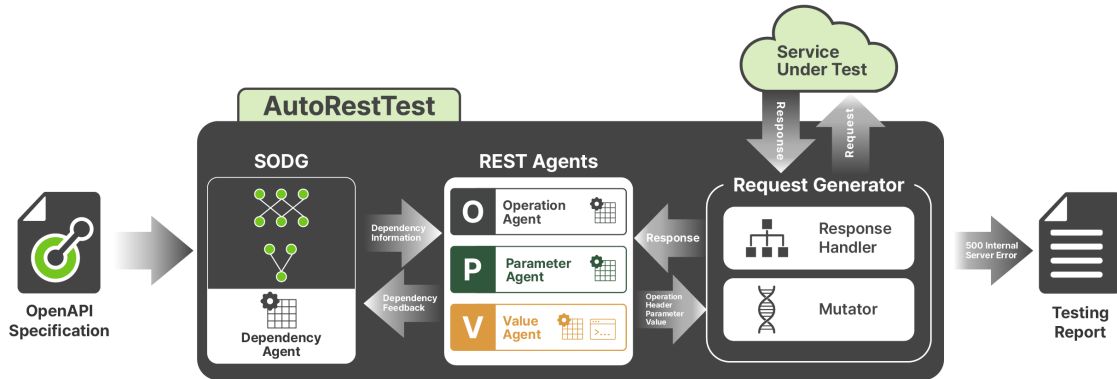


Figure 8.2: Overview of AutoRestTest approach.

Initialization Phase

The process begins with parsing the OpenAPI specification to extract endpoint information, parameters, and request/response schemas. Using the parsed specification, the dependency agent constructs the SPDG as a directed graph in which nodes represent API operations and edges represent potential dependencies between operations. Each weighted edge $e = (a, b)$ indicates that operation b provides values that can be used by operation a (i.e., a depends on b), with the edge weight (a value between 0 and 1) representing the semantic similarity between the operations' inputs and outputs (see Section subsection 8.3.3). These initial dependencies are later validated and refined through the testing process based on actual server responses.

The REST agents (operation, parameter, value, and dependency agents) are then initialized with their respective Q-tables. Each agent serves a specific purpose in the testing process: the operation agent selects API operations to test, the parameter agent determines parameter combinations, the value agent generates appropriate parameter values for each parameter, and the dependency agent manages operation dependencies from the SPDG.

Testing Execution Phase

The testing execution follows an iterative process. In each iteration, the operation agent first selects the next API operation based on its learned Q-values and exploration strategy. Next, the parameter agent determines which parameters to include, considering both required and optional parameters from the specification. The value agent then generates parameter values using dependencies identified by the dependency agent, LLM-generated values, or default assignments for basic parameter types. Using the SPDG, the dependency agent identifies any dependencies between the selected parameters and those used in previous operations. Finally, the request generator constructs the API request, with the mutator component modifying 20% of requests to test error handling and trigger potential 500 response codes, similar to prior work [15].

Once the request is sent to the Service Under Test (SUT) and a response is received, the response is used to update the Q-tables of all agents through reinforcement learning, refine SPDG dependencies, and store any 500 responses for the final testing report. This cycle continues until the testing time budget is exhausted. The SPDG refinement process involves increasing or decreasing edge weights, driven by rewards and penalties for dependencies, based on server responses—successful dependencies (validated by 2xx response codes) increase edge weights, whereas failed dependencies reduce edge weights, with heavily penalized edges being effectively removed from consideration. This continuous refinement helps ensure the accuracy of the dependency graph over time.

8.3.2 Q-Learning and Agent Policy

Both the SPDG and REST agent modules use the Q-learning algorithm [42] with value decomposition within their respective agents. During initialization, each agent creates a Q-table data structure that maps available actions to their expected cumulative rewards. When request generation begins, AutoRestTest addresses the two primary components of Q-learning—action selection and policy optimization—to facilitate effective communica-

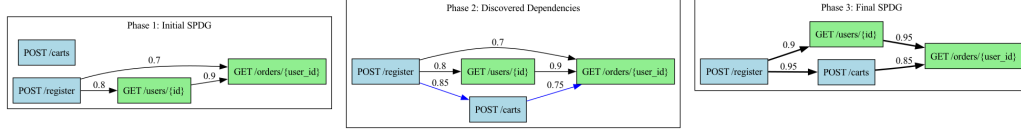


Figure 8.3: Illustration of SPDG construction and refinement.

tion between agents.

Action Selection

During action selection, agents independently choose between exploiting their best-known option or exploring new options randomly. To balance these choices, all agents follow an epsilon-greedy strategy: with probability ϵ , the agent selects a random action (exploration), and with probability $1 - \epsilon$, it selects the action with the highest Q-value (exploitation), likely yielding the best results.

AutoRestTest utilizes epsilon-decay to guarantee all actions are adequately explored in its initial stages. Starting with an epsilon value of 1.0, this value decreases linearly to 0.1 over the duration of the tool’s operation. This strategy, commonly used in practice, has been shown to improve performance by balancing exploration and exploitation [185].

Policy Optimization

After receiving a response from the server, agents update their Q-table values using the temporal-difference update rule of the Q-learning algorithm, derived from the Bellman equation [41]. This update aims to maximize the expected cumulative reward for each action taken. The Q-learning update rule is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta \quad (8.1)$$

where δ represents the temporal-difference error, calculated as:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a) \quad (8.2)$$

where α is the learning rate, γ is the discount factor, r is the received reward, s is the current state, s' is the next state, a is the current action, and a' is an action in state s' .

Given the complexity of the multi-agent environment, AutoRestTest leverages value decomposition to optimize the joint cumulative reward, which has shown improvements for policy acquisition over independent learning. This approach assumes that the joint Q-value can be decomposed additively as follows [181]:

$$Q(s, a) = \sum_i^n Q_i(s, a_i) \quad (8.3)$$

where Q_i and a_i represent the Q-value and action for agent i in shared state s .

The temporal-difference error, which measures the difference between current Q-values and the optimal target Q-values, can be redefined using value decomposition to reflect the displacement from the optimal joint Q-value:

$$\delta = r + \gamma \max_{a'} \sum_i^n Q_i(s', a'_i) - \sum_i^n Q_i(s, a_i) \quad (8.4)$$

Using this redefined temporal-difference error, each agent updates its Q-table to converge towards the optimal joint Q-value, as depicted in the following temporal-difference equation:

$$Q_i(s, a_i) \leftarrow Q_i(s, a_i) + \alpha \left[r + \gamma \max_{a'} \sum_{i=1}^n Q_i(s', a'_i) - \sum_{i=1}^n Q_i(s, a_i) \right] \quad (8.5)$$

This value decomposition approach enables each agent to select actions independently while maintaining centralized policy updates, simplifying the implementation while enhancing coordination across agents [181].

For reward delegation, the dependency, value, and parameter agents are optimized to

reward actions that generate 2xx status codes, whereas the operation agent rewards the selection of operations that generate 4xx and 5xx status codes. This balance in behavior is intended to maximize coverage by encouraging repeated attempts at creating successful requests for operations that frequently yield 4xx and 5xx status codes. For the hyperparameters α and γ , like ARAT-RL [15] and other related work [186, 187], I use values 0.1 and 0.9, respectively.

8.3.3 Semantic Property Dependency Graph

The construction of the SPDG begins with parsing the OpenAPI specification to extract information about API endpoints, parameters, and request/response schemas. As shown in Figure 8.3, the SPDG is initialized as a directed graph where nodes represent API operations and edges represent potential dependencies between operations based on semantic similarities.

The initialization process involves two main steps. First, for each operation in the API specification, I create a node containing the operation’s ID, parameters, and response schemas. Then, I identify potential dependencies between operations by computing semantic similarities between their parameter names (inputs) and response field names (outputs) using cosine similarity with pre-trained word embeddings (e.g., GloVe [188]). When comparing two operations, if their similarity score exceeds 0.7,¹ I create an edge between them with the similarity score as its weight. To ensure all operations have some potential dependencies to explore, if an operation has no edges with scores above the threshold, I connect it to the five most similar operations.²

Phase 1 of Figure 8.3 provides an example of the initial SPDG generated by the technique. For example, the edge between GET /users/id and GET /orders/user_id has a weight of 0.9 due to the high similarity between the names of the inputs and outputs of these two operations.

¹I selected this value based on previous studies [189, 144].

²Top-five is a common threshold in top-k similarity matching [190, 191, 192].

Dependency Agent

The dependency agent manages and uses the dependencies between operations captured in the SPDG. It uses a Q-table to represent the validity of these dependencies, operating similarly to a weighted graph, where edges are assigned values that reflect confidence in each dependency. Specifically, the Q-table encodes edges from the SPDG, categorizing dependencies by parameter type (query or body) and target (parameters, body, or response). Higher Q-values on an edge indicate greater confidence in the reliability of that operation dependency, causing the agent to prioritize these relationships on future requests.

For each parameter and body property in a selected operation, the dependency agent refers to its Q-table to identify a dependent parameter, body, or response, as well as the associated operation ID. For example, as illustrated in Phase 3 of Figure 8.3, when testing `GET /orders/user_id`, the agent might use the value for `user_id` from a successful `GET /users/id` response, reflected by the strengthened edge weight of 0.95. The dependency agent consults AutoRestTest's tables storing successful parameters, request body properties, and decomposed responses to ensure that the selected dependency has available values. AutoRestTest recursively deconstructs response objects, allowing the dependency agent to access nested properties within response collections. Required parameters without available dependencies are fulfilled using value mappings provided by the value agent.

Although the SPDG significantly reduces the search space for operation dependencies, the reliance on semantic similarity may overlook potential candidates. To address this, the dependency agent permits random dependency queries during exploration. As shown in Phase 2 of Figure 8.3, if a random dependency successfully generates a valid input, AutoRestTest identifies the contributing factor and adds this new edge to the SPDG. For instance, during testing, the agent might discover that `POST /carts` returns a `cart_id` that can be used as input for `GET /orders/user_id`, leading to a new edge with weight 0.75. Similarly, when the specification contains undocumented response values (e.g., if `POST /register` returns additional user-related fields not specified in the OpenAPI document),

the dependency agent evaluates these new properties for potential dependencies with other operations, as demonstrated by the edge between `POST /register` and `POST /carts` with weight 0.85 in Phase 2.

8.3.4 REST Agents

Operation Agent

This agent is tasked with selecting the next API operation to test. It employs reinforcement learning to prioritize operations that are likely to yield meaningful test results based on prior experiences. This agent uses a simplified state model that only tracks whether an operation is available for selection. The agent's action space encompasses all possible operations in the API specification. Each operation's Q-value is initialized to 0 and is updated based on response codes from the server. The Q-table for the operation agent stores cumulative rewards representing the proportion of unsuccessful requests for each operation in the provided service, as discussed in greater detail in the next paragraph.

The operation agent acts as the forerunner of the testing process, selecting an operation that dictates the state of the remaining agents. While the remaining agents coordinate to generate valid test cases for a given operation, the operation agent is tasked with identifying unsuccessful operations for retesting. Consequently, while the remaining agents update their Q-value using the value decomposition temporal-difference equation (Equation Equation 8.5 in Section subsection 8.3.2), the operation agent updates its Q-values independently using a structured reward system: +2 for server errors (5xx), +1 for client errors (4xx, excluding 401 and 405), -1 for successful responses (2xx), -3 for authentication failures (401), and -10 for invalid methods (405). This reward structure encourages the agent to prioritize exploring problematic endpoints while severely penalizing systematically invalid requests and mildly penalizing endpoints with consistently successful requests.

As an example, consider the customer registration endpoint in the Market API (Figure 8.1). The Q-value for the endpoint is initially 0. After initial attempts at processing the

operation fail, the Q-table value might increase (e.g., to 1), prompting the agent to prioritize further testing on this endpoint. Conversely, a successful test case would decrease the Q-value, directing the agent to explore other (challenging) endpoints.

Parameter Agent

The parameter agent is responsible for selecting parameters for the chosen API operation. It ensures that parameters used across requests are both valid and varied, covering a range of testing scenarios while addressing inter-parameter dependencies. For each operation, the parameter agent initializes a state containing the operation ID, available parameters, and required parameters, and defines its action space as possible parameter combinations. The Q-values associated with each state-action pair are initialized to 0.

Consider again the Market API's customer registration endpoint shown in Figure 8.1. The parameter agent initializes with the following state: $s = \{\text{createCustomerUsingPOST}, [\text{email}, \text{name}, \text{password}, \text{links}], [\text{email}, \text{name}, \text{password}]\}$, where the first list contains all available parameters from the request body schema, whereas the second list contains only the required parameters according to the endpoint's specification. The agent initializes a Q-table for this operation, mapping various parameter combinations to Q-values.

This setup ensures that different parameter combinations (limited to 10 by default, to account for space restrictions) are sufficiently represented in the agent's action space. The agent updates its Q-values using the value decomposition temporal-difference equation (Equation 8.5 in Section subsection 8.3.2) and the following reward structure: -1 for server errors (5xx), -2 for client errors (4xx), and +2 for successful responses (2xx).

Importantly, unused parameter combinations receive a neutral update (effectively 0) in the Q-learning process, maintaining their initial Q-values. These unused combinations are prioritized over combinations with negative rewards and deprioritized relative to those with positive rewards. For example, for the registration endpoint, if the parameter combination (`email`, `name`, `password`) consistently yields positive rewards, the unused combination

(`email`, `name`, `password`, `links`) retains its initial Q-value and may be selected during exploration to test scenarios with optional parameters. Suppose that the Q-values for these parameter combinations evolved to 0.8 and 0.3, respectively. This would suggest that the inclusion of the `links` parameter is problematic and result in a lower Q-value for the configuration with all parameters.

Through this reward scheme, `AutoRestTest` effectively identifies valid parameter sets and addresses challenges related to undocumented inter-parameter dependency requirements, particularly in complex scenarios such as user registration, where certain parameters must be present and correctly formatted.

Value Agent

This agent is responsible for generating and assigning values to parameters selected by the parameter agent. For each parameter of an operation, it maintains a state containing the operation ID, parameter name, parameter type, and OpenAPI schema constraints, with its actions corresponding to possible data sources for parameter value assignment. The Q-values for each state-action pair are initialized to 0.

Consider the Market API's customer registration endpoint in Figure 8.1. The value agent initializes the following states for `email`, `name`, and `password` parameters:

- $s = \{\text{createCustomerUsingPOST}, \text{email}, \text{string}, \{\text{pattern: } ^{[\backslash w-]}+(\backslash.[\backslash w-])^* @([\backslash w-]+\backslash.)+[a-zA-Z]+\$\}\}$
- $s = \{\text{createCustomerUsingPOST}, \text{name}, \text{string}, \{\text{pattern: } ^{[\backslash pL ' -]}+\backslash\$, \text{maxLength: } 50\}\}$
- $\{\text{createCustomerUsingPOST}, \text{password}, \text{string}, \{\text{pattern: } ^{[a-zA-Z0-9]}+\$, \text{minLength: } 6, \text{maxLength: } 50\}\}$

To generate a diverse set of inputs, the value agent can select inputs from the following data sources:

- *Operation Dependency Values*: When selected, the value agent collaborates with the dependency agent to map dependent values to the selected parameter. For example, the registration endpoint might reuse email addresses from prior successful registrations to test duplicate user scenarios.
- *LLM Values*: For this data source, the value agent creates (or parses if already created) values using few-shot prompting with LLMs [25].³ For instance, the LLM might generate “john.doe@example.com” as the input value for the `email` parameter based on the specified pattern.
- *Random Values*: When this option is selected, the value agent generates random values based on the type of the selected parameter. For example, it may create a random sequence of 1-50 characters for strings, a random number between -1024 and 1024 for integers, and a random true/false value for boolean types.

Once a request is completed with values from the chosen data source, the agent updates its Q-values based on the temporal-difference equation (Equation Equation 8.5 in Section subsection 8.3.2), using the same reward strategy as the parameter agent (§subsection 8.3.4) to refine its value generation.

For the registration endpoint, for instance, the Q-values across parameters show an average of 0.5 for LLM values, 0.2 for random values, and -0.7 for operation dependency values. In analyzing these Q-values, I observe that because the registration endpoint is required for account creation, it is less likely to derive values from operation dependencies, which results in a lower Q-value for the dependency source. Although random values are effective for simpler parameters, such as `name`, the LLM-generated values perform better for pattern-constrained fields, such as `email` and `password`.

³In this work, I used GPT-3.5 Turbo with a temperature setting of 0.8 because of its known performance in REST API contexts [160].

8.3.5 Request Generator

The request generator constructs and dispatches API requests to the SUT. It works closely with the REST agents to ensure that the generated requests are both effective and comprehensive. Upon receiving responses from the SUT, the response handler processes these results and provides feedback to the REST agents, allowing refinement of future requests.

The mutator's purpose is to generate invalid requests to uncover unexpected behaviors (e.g., 500 responses). This is a crucial part of REST API testing frameworks, as state-of-the-art tools employ similar strategies such as mutating parameter types, values, and headers (e.g., using an invalid content type in the header). The mutator follows these conventions and mutates 20% of the generated requests, a strategy used by the most recent tool [15].

The request generator interacts with the REST agents to construct API requests, relying on them for detailed information about the operation to test, the parameters to use, and appropriate values for those parameters. Specifically:

- The operation agent selects the API operation to test.
- The parameter agent identifies and optimizes the parameters for the chosen operation.
- The value agent generates realistic and effective values for the parameters.

Using this information, the request generator constructs a complete API request. The request generator then dispatches the request to the SUT, which processes the request and returns a response. The response handler analyzes this response to detect any errors or unexpected behaviors. Insights from these analyses are fed back to both the REST agents and the dependency agent, allowing them to refine their strategies for future requests.

The interactions between the dependency module, the REST agents, the request generator, and the SUT establish a robust feedback loop that enhances the overall effectiveness

of the testing process. This collaborative approach ensures that the generated requests are not only comprehensive but also tailored to uncover potential issues within REST APIs.

8.4 Evaluation

In this section, I present the results of empirical studies conducted to assess AutoRestTest. This evaluation aims to address the following research questions:

1. **RQ1:** How does AutoRestTest compare with state-of-the-art REST API testing tools in terms of code coverage and operation coverage achieved?
2. **RQ2:** In terms of error detection, how does AutoRestTest perform in triggering 500 (Internal Server Error) responses compared to state-of-the-art REST API testing tools?
3. **RQ3:** How do the main components of AutoRestTest (MARL, SPDG, and LLM-based input generation) contribute to its overall performance?

8.4.1 Experiment Setup

I conducted the experiments on two cloud VMs, each equipped with a 48-core Intel(R) Xeon(R) Platinum 8260 processor with 128 GB RAM. To ensure consistent test conditions, I restarted the services and restored their databases in each testing session to eliminate potential state dependency effects across sessions. I used the default configuration and database settings for each service. I allocated dedicated resources to each service and testing tool, running them sequentially to prevent interference. Throughout the experiments, I closely monitored CPU and memory usage to ensure optimal performance without encountering resource constraints.

For the evaluation, I relied on the same set of REST API testing tools and services used by ARAT-RL [15]. Accordingly, I compared AutoRestTest with ARAT-RL [15], EvoMaster [45], MoRest [17], and RESTler [9]. Specifically, I used the latest released version

Table 8.1: REST services used in the evaluation.

REST Service	Lines of Code	#Operations
Features Service	1688	18
Language Tool	113170	2
Rest Countries	1619	22
Genome Nexus	22143	23
Person Controller	601	12
User Management	2805	22
Market	7945	13
Project Tracking System	3784	67
YouTube-Mock	2422	1
FDIC	–	6
Spotify	–	12
OhSome API	–	122

or the latest commit when a release was unavailable: RESTler v9.2.4, EvoMaster v3.0.0, ARAT-RL v0.1, MoRest (obtained directly from the authors).

The ARAT-RL benchmark dataset has 10 RESTful services. In addition to these services, I included the services from the RESTGPT study [160]. Out of the total 16 services, I excluded SCS and NCS because they were written by EvoMaster’s authors, and I aimed to avoid potential bias. I also excluded OCVN due to authentication issues. Lastly, I excluded OMDB, which is a toy online service with only one API operation that all testing tools can process in a second. Ultimately, I used 12 services: Features Service, Language Tool, REST Countries, Genome Nexus, Person Controller, User Management Microservice, Market Service, Project Tracking System, OhSome, YouTube-Mock, and Spotify. Table 8.1 lists the open-source services along with the lines of code and the number of API operations in each service.

For a fair comparison, because the tool utilizes LLM calls, I used the enhanced specification generated by RESTGPT, which adds realistic testing inputs to the specification using LLMs. Moreover, I used GPT-3.5-Turbo, as RESTGPT utilized this model. Based on a recent survey that describes settings and metrics for REST API testing [163], I used a one-hour time budget with ten repetitions to compute the results. To measure effectiveness and error-finding ability, I used code coverage (open-source services only), number

of successfully processed operations in the specification, and number of 500 status codes, which are the most popular metrics in the literature. To collect code coverage, I used Jacoco [114]. For the number of processed operations, I used the script from the NLP2REST repository [131]. For the number of 500 status codes, I used the script available in the ARAT-RL repository [150]. This script collects 500 status codes by tracking the HTTP responses, and removes the duplicated 500 codes using the server response message for each operation.



Figure 8.4: Comparison of code coverage metrics across tools and services: line, branch, and method coverage.

8.4.2 RQ1: Effectiveness

The effectiveness of AutoRestTest is evaluated based on its ability to comprehensively cover more code compared to other tools. Figure 8.4 illustrates the line, branch, and method coverage achieved by each testing tool on the nine open-source services in this benchmark; additionally, it shows the average coverage across these APIs.

Table 8.2: Number of operations exercised.

	AutoRestTest	ARAT-RL	EvoMaster	MoRest	RESTler
FDIC	6	6	6	6	6
OhSome	12	0	0	0	0
Spotify	7	5	4	4	3
Total	25	11	10	10	9

As shown in Figure 8.4, AutoRestTest outperformed the other tools in terms of method coverage, achieving 58.3% coverage on average, compared to ARAT-RL (42.1%), EvoMaster (43.1%), MoRest (31.5%), and RESTler (34.7%). This represents a significant coverage gain ranging from 15.2 to 26.8 percentage points. Similarly, AutoRestTest achieved higher line coverage, 58.3% on average, compared to the other tools, which achieved 44%, 44.1%, 33.4%, and 34.6%, respectively. Finally, in terms of branch coverage, AutoRestTest again outperformed the other tools, achieving 32.1% coverage on average compared to the other tools, which achieved 19.8%, 20.5%, 12.7%, and 11.4%, respectively.

In this evaluation, I also measured the number of processed operations for online services for which source code is unavailable: OhSome and Spotify. Table 8.2 presents these results, which highlight AutoRestTest’s ability to handle a larger number of operations. Specifically, AutoRestTest exercised 25 operations in total, compared to ARAT-RL, EvoMaster, MoRest, and RESTler, which processed 11, 10, 10, and 9 operations, respectively. These results on achieved code coverage and successfully exercised operations demonstrate AutoRestTest’s effectiveness on a range of different REST APIs and how it improves on the state of the art.

In most cases, there were notable performance gains in Genome Nexus, Person Controller, User Management, Market, YouTube, OhSome, and Spotify. Conversely, for Features Service, REST Countries, and Project Tracking System, AutoRestTest’s results did not show much difference compared to the second-best performing tool. These four services have a notable characteristic in common: the number of input parameters in their APIs is mostly 1 to 2, and the services are therefore easier to test. This result shows that Au-

Table 8.3: Service failures triggered (500 response codes).

REST APIs	AutoRestTest	ARAT-RL	EvoMaster	MoRest	RESTler
Features Service	1	1	1	1	1
Language Tool	1	1	1	0	0
REST Countries	1	1	1	1	1
Genome Nexus	1	1	0	1	0
Person Contoller	8	8	8	8	3
User Management	1	1	1	1	1
Market	1	1	1	1	1
Project Tracking System	1	1	1	1	1
YouTube	1	1	1	1	1
FDIC	6	6	6	6	6
OhSome	20	12	0	0	0
Spotify	1	0	0	0	0
Total	42	33	20	20	14

toRestTest can effectively explore REST APIs, especially for services with a large search space.

AutoRestTest achieves considerable gains in code coverage, with method coverage increasing between 15.2 to 26.8 percentage points, line coverage between 14.2 and 24.8 percentage points, and branch coverage between 11.6 and 20.7 percentage points compared to the other tools considered. The improvement in performance is particularly noticeable on large and complex services with many input parameters.

8.4.3 RQ2: Fault Detection Capability

I evaluated the fault detection capability of AutoRestTest by counting how many 500 Internal Server Errors it identified. Table 8.3 shows the number of such errors detected by AutoRestTest, ARAT-RL, EvoMaster, MoRest, and RESTler. As the data in the table show, AutoRestTest detected a total of 42 500 Internal Server Errors across the evaluated REST APIs, far outperforming the other tools on this metric. ARAT-RL detected 33 errors, EvoMaster and MoRest detected 20 errors each, and RESTler detected 14 errors.

Specifically, AutoRestTest identified significantly more errors in the OhSome service (20 errors) compared to ARAT-RL (12 errors), with none detected by EvoMaster, MoRest,

or RESTler. Additionally, AutoRestTest was the only tool to detect an error in the Spotify service. It is important to note that both OhSome and Spotify are active services; Spotify, for example, has 615 million users, and the OhSome service has recent GitHub commits. I reported the detected issues, and the OhSome errors were accepted, whereas I am still waiting for Spotify's response [182, 183]. This improvement in fault detection is somehow expected, as AutoRestTest achieves the highest coverage among the other tools, which is strongly correlated with fault-finding ability in REST API testing [19].

To illustrate the utility of AutoRestTest's specific components in fault detection on a specific example, consider the following sequence of operations in the OhSome service, which shows the capabilities of the SPDG. AutoRestTest begins by successfully querying the `POST /elements/area/ratio` endpoint from the OhSome service with its `filter2` parameter assigned to `node:relation`. Subsequently, AutoRestTest targets the `GET /users/count/groupBy/key` endpoint, where the dependency agent applies the SPDG's semantically-created dependency edges to identify a potential connection between the `filter` parameter of the new operation and the `filter2` parameter of the previous operation. When the dependency agent reuses the `node:relation` value from the previously successful `filter2` parameter in the new request, the SPDG uncovers an unexpected 500 Internal Server Error. Typically, an invalid `filter` value would trigger a client error, but this server-side error indicates that the SPDG identified a deeper, unanticipated fault in the server's value handling. The other tools overlook the correlation between the two parameters due to either the minor naming differences or improper dependency modeling, and fail to expose this error.

For another example, AutoRestTest shows the effectiveness of its LLM value generation in the interactions with the Spotify API. The `GET /playlists/playlist_id/tracks` operation in Spotify's API requires specific knowledge regarding Spotify's `playlist_id` formation. Spotify generates Spotify IDs for playlists that are typically 22 characters long with constraints on the permitted letters and patterns. Where many tools fail

Table 8.4: Code coverage achieved by different tool variants.

	Method	Line	Branch
AutoRestTest	58.3%	32.1%	58.3%
1. Without LLM	47.4% (-10.9%)	19.3% (-12.8%)	45.8% (-12.5%)
2. Without Learning	45.6% (-12.7%)	18.2% (-13.9%)	45.8% (-12.5%)
3. Without SPDG	46.7% (-11.6%)	18.7% (-13.4%)	47.6% (-10.7%)

to create valid IDs, AutoRestTest’s value agent leverages its LLM to generate valid Spotify IDs for playlists. AutoRestTest is thus able to successfully query the `GET /playlists/playlist_id/tracks` operation, with rippling effects across the service. For instance, after retrieving the International Standard Recording Code (ISRC) from the playlist’s tracks, AutoRestTest’s mutator randomly selects an ISRC to use as the `user_id` in the subsequent `GET /users/user_id/playlists` operation. This sequence reveals a hidden dependency conflict that results in a 500 Internal Server Error. Other tools fail to exercise the `GET /playlists/playlist_id/tracks` operation entirely and are hence unable to locate this error.

AutoRestTest outperforms the other tools in terms of fault detection capability. It identifies a total of 42 instances of 500 Internal Server Errors, whereas ARAT-RL, EvoMaster, MoRest, and RESTler detected 33, 20, 20, and 14 errors, respectively.

8.4.4 RQ3: Ablation Study

To understand the contribution of each component in AutoRestTest, I conducted an ablation study in which I removed specific elements of the approach: the LLM-based input generation, the agent learning step in MARL, and the SPDG. Because this tool heavily depends on agents, it was not feasible to create a reasonable tool without MARL entirely. Therefore, instead of removing all the agents, I only removed the temporal-difference Q-learning from the agents. Table 8.4 presents the impact of these removals on method, line, and branch coverage.

Removing the temporal-difference Q-learning leads to the most significant decrease in

overall metrics, dropping the coverage rates to 45.6%, 18.2%, and 45.9% for method, line, and branch coverage. The learning step's contribution is crucial in optimizing the testing process through strategic exploration and exploitation of testing paths. Without the multi-agent learning step, the tool repeated the same requests and failed to properly update its agents with feedback.

The removal of the SPDG has the next most significant impact in terms of method and line coverage, dropping the performance significantly to 46.7%, 18.7%, and 47.6% for method, line, and branch coverage. This result indicates that the SPDG plays a critical role in identifying dependencies and guiding test generation by reducing the search space, thus helping identify the dependent API operations.

Removing the LLM alone also leads to a substantial decrease in performance, with method, line, and branch coverage dropping to 47.4%, 19.3%, and 45.8%. The primary reason for this difference is the LLM's ability to generate diverse and appropriate test inputs. These inputs are essential for exercising the API, as they help uncover various parameter and operation dependencies. Furthermore, operation and parameter dependencies cannot be accurately identified without resolving parameter constraints when they exist.

Notably, without the SPDG, AutoRestTest exercised only 5 operations for Spotify, whereas with the SPDG, it consistently covered 7 operations.

The ablation study shows that each component of AutoRestTest (LLM, MARL, and SPDG) contributes considerably to the effectiveness of the approach, and removing any component drops the coverage significantly. The decrease in method, line, and branch coverage ranges, in percentage points, between 10.9 and 12.7, 12.8 and 13.9, and 10.7 and 12.5, indicating that each component plays an important role.

8.4.5 Threats to Validity

Like for any empirical study, there are potential threats to the validity of the results. Regarding construct validity, the use of ChatGPT-3.5-Turbo as the LLM component [24] introduces potential data leakage, as it may have been trained on API-related content, potentially affecting results. While the ablation study demonstrates the LLM’s importance, this limitation should be considered when interpreting the findings. Additionally, technical choices like the 20% mutation rate [15] and ChatGPT’s default parameters [160] may affect result comparability.

REST APIs’ inherently flaky behaviors (e.g., due to network issues) introduces possible threats of internal validity. To mitigate this issue, I performed multiple test runs and averaged the results. I also carefully inspected the code and results to mitigate the risk of implementation errors.

The uneven quality of OpenAPI specifications [193] can also affect the validity of the results. While this is an inherent and somehow unavoidable issue, I have left to future work the investigation of the impact of specification completeness and accuracy on AutoRestTest’s performance.

Finally, the selection of REST APIs benchmarks can affect external validity. While I used a diverse set of real-world APIs, AutoRestTest may perform differently on different benchmarks. The availability of the dataset and code will allow other researchers to validate and extend this evaluation.

8.5 Conclusion

In this paper, I introduced AutoRestTest, a new technique that leverages multi-agent reinforcement learning, the semantic property dependency graph, and large language models to enhance REST API testing. By optimizing specialized agents for dependencies, operations, parameters, and values, AutoRestTest addresses the limitations of existing techniques and

tools. The evaluation on 12 state-of-the-art REST services shows that AutoRestTest can significantly outperform leading REST API testing tools in terms of code coverage, operation coverage, and fault detection. Furthermore, the ablation study confirms the individual contributions of the MARL, LLMs, and SPDG components to the tool’s effectiveness.

CHAPTER 9

CONCLUSION AND FUTURE WORK

This dissertation has presented an investigation of machine learning techniques for enhancing black-box REST API testing. Through a progression of approaches, this research has addressed critical limitations in existing testing tools, particularly in areas of coverage, fault detection, and intelligent test input generation. The work is motivated by the increasing prevalence and importance of RESTful APIs in modern software architectures, coupled with the pressing need for more robust and sophisticated testing mechanisms.

9.1 Summary of Contributions

This research has made three main categories of contributions to the field of REST API testing:

9.1.1 Empirical Understanding of API Testing Limitations

The research began with a thorough analysis of ten state-of-the-art REST API testing tools across 20 RESTful web services, revealing significant gaps in current testing capabilities. This empirical study provided valuable insights into specific challenges:

- The difficulty of generating valid input values for domain-specific parameters
- The limited ability to detect and navigate complex dependencies between operations and parameters
- The inefficient exploration of the vast API behavior space

9.1.2 Specification Enhancement Approaches

To address limitations in existing API specifications, I developed two complementary approaches:

- **NLP2REST**: This pioneering approach leveraged traditional NLP techniques to extract valuable information from human-readable parts of OpenAPI specifications, significantly enhancing the effectiveness of existing testing tools by bridging the gap between documentation and machine-readable specifications.
- **RestGPT**: Building upon NLP2REST, this work introduced the use of Large Language Models for specification enhancement, demonstrating superior performance in interpreting context, semantics, and generating relevant test inputs without requiring expensive validation processes.

These approaches enable existing testing tools to perform more effectively without modifying their core algorithms, demonstrating the value of enriching specifications with information extracted from natural language documentation.

9.1.3 Advanced Testing Methodologies

Moving beyond specification enhancement, I developed three increasingly sophisticated approaches to fundamentally improve the testing methodology itself:

- **ARAT-RL**: This innovative approach employed Q-learning to dynamically prioritize API operations and parameters, significantly outperforming state-of-the-art tools across multiple metrics by intelligently navigating the API dependency space.
- **LlamaRestTest**: This contribution showcased the effectiveness of fine-tuned and quantized small language models in REST API testing, achieving a balance between model size and accuracy while maintaining high performance in resource-aware environments.

- **AutoRestTest:** The culmination of this research integrated Multi-Agent Reinforcement Learning with a Semantic Property Dependency Graph and Large Language Models, demonstrating how specialized agents can collaborate to address the complex challenge of REST API testing.

This progression demonstrates a systematic exploration of how different machine learning paradigms can address increasingly complex aspects of the API testing challenge, with each approach building upon insights gained from previous ones.

9.2 Addressing the Research Questions

9.2.1 Main Research Question

Can machine learning techniques and language models enhance the effectiveness of black-box REST API testing by increasing code and API coverage, improving fault detection, and addressing limitations of traditional approaches?

This dissertation has demonstrated that machine learning techniques can significantly enhance black-box REST API testing through multiple complementary approaches. Beginning with natural language processing techniques to extract valuable information from API documentation, progressing to large language models for more accurate rule extraction and value generation, and culminating in sophisticated reinforcement learning and multi-agent systems, this research has shown a clear progression of improvements in testing effectiveness.

The empirical results across all implemented approaches consistently demonstrated significant enhancements in three key areas:

1. **Increased Coverage:** The machine learning techniques developed in this work consistently achieved higher code coverage (branch, line, and method) and API operation coverage compared to traditional approaches.

2. **Improved Fault Detection:** All approaches demonstrated enhanced ability to detect unique faults, with the later approaches (ARAT-RL, LlamaRestTest, and AutoRestTest) showing particular strength in triggering hard-to-reach error conditions. In several cases, these techniques were able to identify previously unknown issues in production APIs that traditional tools had failed to detect.
3. **Addressing Key Limitations:** The machine learning approaches directly addressed the three main limitations identified in the empirical study:
 - Generating valid input values for domain-specific parameters through LLMs and specialized fine-tuned models
 - Detecting and navigating complex dependencies through reinforcement learning and semantic dependency graphs
 - Efficiently exploring vast API behavior spaces through adaptive, experience-driven strategies

Furthermore, this research demonstrated that different machine learning paradigms offer distinct advantages for API testing. NLP and LLMs excel at extracting knowledge from textual descriptions, reinforcement learning provides adaptive exploration strategies, and multi-agent systems allow for specialized coordination of testing activities. By systematically investigating these approaches, this dissertation has shown that machine learning can transform REST API testing from a primarily static, specification-driven activity to a dynamic, intelligent process that learns and adapts based on API behavior.

9.2.2 Specific Research Questions

RQ1: Can natural language processing extract useful information from OpenAPI Specification to improve REST API testing?

Through the development of NLP2REST (Chapter 4), this research demonstrated that traditional NLP techniques can effectively extract valuable information from human-

readable API documentation that is typically overlooked by automated testing tools. By employing a custom Word2Vec model and constituency parse trees, NLP2REST successfully identified parameter constraints, dependencies, and examples from textual descriptions. The extracted information was validated through both static analysis and dynamic checking, resulting in enhanced OpenAPI specifications that led to substantial improvements in test generation, with measurable increases in successful API requests, code coverage, and fault detection.

RQ2: Can large language models improve test input generation and parameter constraint identification in REST API testing?

RestGPT (Chapter 5) answered this question by demonstrating that Large Language Models can significantly outperform traditional NLP approaches in both rule extraction and test input generation. Through carefully crafted prompts that include guidelines, specific cases, and output configurations, LLMs showed superior ability to understand context and nuance in natural language descriptions. This enabled RestGPT to extract operational constraints, parameter constraints, type/format information, and examples with higher precision than previous approaches, without requiring expensive validation processes. The results showed marked improvements in generating valid inputs that respect domain-specific constraints, leading to more effective test cases.

RQ3: Can reinforcement learning be applied to create adaptive testing strategies that efficiently explore the behavior space of REST APIs?

ARAT-RL (Chapter 6) addressed this question by implementing a Q-learning algorithm that dynamically adjusts the importance of API operations and parameters based on feedback received from the API. This adaptive approach demonstrated how reinforcement learning can optimize the testing process through experience-driven exploration, focusing on areas of the API more likely to increase code coverage. The dynamic construction of key-value pairs from both request and response data enabled ARAT-RL to identify hidden dependencies that may not be evident from specifications alone. Experimental evaluations

confirmed that this reinforcement learning strategy consistently outperformed state-of-the-art tools across various metrics, including branch, line, and method coverage, as well as fault detection.

RQ4: What benefits can specialized, fine-tuned language models bring to REST API testing in terms of both effectiveness and computational efficiency?

LlamaRestTest (Chapter 7) answered this question by demonstrating that fine-tuned and quantized small language models can achieve state-of-the-art performance in API testing while maintaining computational efficiency. By developing two specialized models—one for identifying inter-parameter dependencies and another for generating input values—LlamaRestTest achieved an optimal balance between effectiveness and resource consumption. The research showed that these compact models, when properly fine-tuned on domain-specific data, can successfully incorporate server feedback into the testing process while requiring significantly fewer computational resources than larger models. Comparative evaluations confirmed substantial improvements in code coverage, operation coverage, and fault detection capability over existing tools.

RQ5: Can a multi-agent approach coordinate specialized testing components to improve the comprehensiveness and effectiveness of REST API testing?

AutoRestTest (Chapter 8) addressed this question by demonstrating the effectiveness of a multi-agent reinforcement learning framework where four specialized agents—API, dependency, parameter, and value—collaborate to optimize API exploration. The research showed that treating REST API testing as a separable problem allows each agent to focus on its specific aspect while maintaining a cohesive overall testing strategy through value decomposition techniques. The integration of a Semantic Property Dependency Graph simplified the search space for dependencies, while LLMs handled domain-specific value restrictions. Experimental evaluations confirmed that this collaborative approach significantly outperformed existing tools, including being the only approach able to trigger certain internal server errors in production systems. This demonstrated that a well-coordinated

multi-agent approach can effectively combine the strengths of different specialized components to achieve more effective API testing.

9.3 Impact and Benefits

The methodologies developed in this research represent significant advancements in the field of automated REST API testing. By systematically addressing key limitations in existing tools, these approaches offer several important benefits:

Improved Test Coverage: The novel techniques presented here consistently demonstrated superior code and operation coverage compared to existing tools, ensuring more effective testing of REST APIs.

Enhanced Fault Detection: Across all methodologies, there was a marked improvement in the ability to detect unique faults and generate fault-inducing requests, contributing to more robust and reliable API implementations.

Intelligent Test Input Generation: The use of NLP, LLMs, and reinforcement learning techniques has improved the quality and relevance of generated test inputs, addressing a critical shortcoming in existing tools.

Efficiency and Adaptability: The adaptive nature of approaches like ARAT-RL and AutoRestTest allows for more efficient testing processes that can dynamically adjust to the specific characteristics of different APIs.

Resource-Aware Solutions: The development of techniques like LlamaRestTest demonstrates that effective testing can be achieved with smaller, more efficient models, making advanced testing capabilities accessible in resource-constrained environments.

9.4 Reflections on Tools and Methodology

Looking back at the progression of tools developed throughout this research, several key insights emerge:

What Worked Well: The incremental approach to tool development proved highly effective, with each new method building upon lessons from previous ones. Starting with simpler NLP techniques before progressing to LLMs and reinforcement learning allowed for systematic identification of strengths and limitations. The multi-agent approach in AutoRestTest successfully combined the strengths of previous approaches while mitigating their individual weaknesses.

Areas for Improvement: If redesigning these tools from the beginning, several adjustments would enhance their effectiveness:

- Earlier integration of domain-specific knowledge into the testing process
- More focus on response validation beyond status codes
- Development of more efficient fine-tuning strategies for specialized models
- Greater emphasis on explainability of test case generation decisions

The evolution from rule-based approaches to learning-based techniques demonstrated the importance of adaptability in API testing. While rule-based systems excelled at handling well-documented constraints, learning-based approaches proved superior for discovering implicit behaviors and adapting to API-specific characteristics.

9.5 Merit of Research

This research makes several valuable contributions to both the academic understanding and practical application of REST API testing:

Bridging Theory and Practice: By systematically applying machine learning techniques to the practical challenge of API testing, this work connects theoretical advances in AI/ML with real-world software engineering needs. The demonstrated improvements in coverage and fault detection provide empirical evidence of the value of these approaches in production environments.

Enabling More Robust Distributed Systems: As software architectures increasingly rely on microservices and distributed components that communicate via REST APIs, the enhanced testing methodologies developed in this research directly contribute to more reliable and robust system implementations. By improving the quality assurance process for APIs, this work helps prevent costly integration issues and runtime failures in production systems.

Setting New Benchmarks: The comprehensive evaluation across multiple APIs and comparison with existing tools establishes new performance benchmarks for REST API testing. These results provide a foundation for future research and give practitioners clear metrics for evaluating testing approaches.

Contributing to Knowledge Transfer: The systematic exploration of how different machine learning paradigms can be applied to software testing contributes to broader knowledge transfer between the AI and software engineering communities, potentially inspiring similar applications in other testing domains.

9.6 Limitations and Challenges

Despite the advances presented in this research, several limitations and challenges should be acknowledged:

Evaluation Scale: While the evaluations included 9-20 REST APIs of varying complexity, this still represents a limited sample of the vast ecosystem of REST APIs in production. The performance on extremely large-scale APIs with thousands of endpoints might differ from the results presented here.

Computational Requirements: Some of the approaches, particularly those leveraging large language models, require significant computational resources for training and, to a lesser extent, inference. Although LlamaRestTest demonstrated progress toward more efficient solutions, further work is needed to make these techniques accessible in broader development environments.

Dynamic API Behavior: APIs that exhibit significant temporal variations in behavior or strong dependencies on external systems pose additional challenges not fully addressed in this research. Testing approaches may need further adaptation to handle such highly dynamic systems effectively.

Ground Truth Limitations: Evaluating fault detection effectiveness is inherently limited by our knowledge of existing faults. While the research identified new issues in several APIs, there may be undiscovered faults that none of the testing approaches detected.

Response Correctness Assessment: The current work primarily focuses on explicit errors (status codes) and coverage metrics rather than deeply validating the correctness of responses that return success codes, which represents an area requiring further investigation.

9.7 Directions for Future Research

While this research has made significant strides in advancing automated REST API testing through a systematic exploration of machine learning techniques, it also opens several promising avenues for future work.

9.7.1 Enhancing REST API Testing Techniques

The current work demonstrates the potential of AI-based techniques for REST API testing, but several important extensions could significantly improve their effectiveness. Future research should focus on improving code coverage and detecting more subtle types of faults that current approaches miss. Particularly important is developing more sophisticated state tracking mechanisms to enable deeper exploration of API behavior paths and creating specialized test oracles that can identify semantic inconsistencies in responses.

A critical limitation in current REST API testing approaches is their focus on HTTP status codes as the primary oracle. Future work should investigate techniques for detecting "silent failures" where services return 200 OK status codes but provide functionally incor-

rect responses. This might involve mining bug repositories to understand the proportion of REST API issues that manifest as incorrect 200 responses versus explicit errors, then developing specialized testing approaches to detect such silent failures.

While our current techniques excel at exploratory fuzzing, extending them to generate functional and regression tests represents a valuable direction. This would require mining API usage patterns from client applications to understand typical usage scenarios and generating parameterized test suites that capture expected behaviors. Additionally, automated debugging and fault localization techniques could help developers diagnose and fix identified issues by tracing faults from API responses back to implementation issues and suggesting potential fixes based on patterns observed in successful API interactions.

9.7.2 Advanced Code Generation from API Specifications

Current OpenAPI generators like Swagger CodeGen produce only basic boilerplate code, leaving developers to implement the actual business logic. Our research provides a foundation for more advanced code generation that could produce substantial portions of API implementations. Unlike basic generators, future techniques could infer business logic from combinations of path specifications, example responses, and schema constraints.

To achieve more complete API implementation generation, OpenAPI specifications would need enrichment with semantic annotations indicating business rules and constraints. Our testing techniques could be leveraged to improve code generation by validating generated implementations and incrementally refining them based on test results. This represents an important synergy between testing and implementation—using automatically generated tests to guide and validate automatically generated implementations.

The techniques presented in this dissertation, which use OpenAPI specifications as a starting point without examining service implementation, could be extended to support sophisticated implementation generation. This would require enhanced specifications that capture value constraints and input-output dependencies between parameters. Such re-

search could dramatically improve developer productivity by generating not just interface definitions but functional implementations that require minimal human intervention.

9.7.3 Cross-Pollination with Unit Testing

The methods presented in this dissertation could be adapted for unit test generation by treating function interfaces analogously to API endpoints, with parameters as inputs and return values as responses. This would involve adapting our dependency inference techniques to detect dependencies between functions and extending our LLM-based input generation approaches to create relevant test inputs based on function signatures and documentation.

A promising research direction is bridging the gap between unit and integration testing. Our approach already targets integration testing of APIs, but these techniques could be extended to generate comprehensive test suites that exercise code at multiple levels of granularity. This would involve connecting unit tests for individual components with integration tests for API endpoints and system tests that exercise complete workflows spanning multiple API calls.

9.7.4 Broader Applications

Beyond the immediate extensions described above, our techniques show potential for application to other API paradigms and testing scenarios. They could be adapted to test GraphQL APIs, gRPC services, SOAP interfaces, and WebSocket-based APIs. The reinforcement learning approaches developed in this work could be particularly valuable for performance testing by modifying reward functions to prioritize resource consumption metrics, while our LLM-based input generation techniques could be specialized for security testing to produce potential attack vectors.

Testing complex stateful systems represents another challenging frontier. Future research could explore how our techniques might be extended to test APIs with intricate state machines and long-running transactions that span multiple API calls. This would require

more sophisticated state modeling and transition inference than current approaches provide.

Conclusion

In conclusion, this dissertation has demonstrated how a systematic progression of machine learning techniques can significantly enhance REST API testing. By methodically addressing different aspects of the testing challenge and building upon insights gained at each step, this work provides a foundation for future research in improving both REST API testing and the broader software development lifecycle.

REFERENCES

- [1] C. Pautasso, O. Zimmermann, and F. Leymann, “Restful web services vs. ”big” web services: Making the right architectural decision,” in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW ’08, Beijing, China: Association for Computing Machinery, 2008, pp. 805–814, ISBN: 9781605580852.
- [2] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs: Services for a Changing World*. O’Reilly Media, Inc., 2013.
- [3] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.
- [4] *Apis.guru api directory*, <https://apis.guru/openapi-directory/>, 2022.
- [5] R Software Inc., *Rapidapi*, 2023.
- [6] OpenAPI, *Openapi standard*, <https://www.openapis.org>, 2023.
- [7] S. Software, *Swagger*, <https://swagger.io/specification/v2/>, 2023.
- [8] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, “Automated black-box testing of nominal and error scenarios in restful apis,” *Software Testing, Verification and Reliability*, vol. 32, Jan. 2022.
- [9] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Restler: Stateful rest api fuzzing,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 748–758.
- [10] S. Karlsson, A. Čaušević, and D. Sundmark, “Quickrest: Property-based test generation of openapi-described restful apis,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 131–141.
- [11] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “Restest: Automated black-box testing of restful web apis,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 682–685, ISBN: 9781450384599.
- [12] S. Karlsson, A. Čaušević, and D. Sundmark, “Automatic property-based testing of graphql apis,” *arXiv preprint arXiv:2012.07380*, 2020.
- [13] Z. Hatfield-Dodds and D. Dygalo, “Deriving semantics-aware fuzzers from web api schemas,” in *Proceedings of the ACM/IEEE 44th International Conference*

- on Software Engineering: Companion Proceedings*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 345–346, ISBN: 9781450392235.
- [14] H. Wu, L. Xu, X. Niu, and C. Nie, “Combinatorial testing of restful apis,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 426–437, ISBN: 9781450392211.
- [15] M. Kim, S. Sinha, and A. Orso, “Adaptive rest api testing with reinforcement learning,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 446–458.
- [16] *Tcases restapi tool*, Accessed: Jun 3, 2022, 2022.
- [17] Y. Liu *et al.*, “Morest: Model-based restful api testing with execution feedback,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1406–1417, ISBN: 9781450392211.
- [18] E. P. Kao, *An introduction to stochastic processes*. Courier Dover Publications, 2019.
- [19] M. Kim, Q. Xin, S. Sinha, and A. 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*, ser. ISSTA 2022, Virtual, South Korea: Association for Computing Machinery, 2022, pp. 289–301, ISBN: 9781450393799.
- [20] M. Zhang and A. Arcuri, “Open problems in fuzzing restful apis: A comparison of tools,” *ACM Trans. Softw. Eng. Methodol.*, May 2023.
- [21] *Restful api modeling language*, Accessed: Jun 3, 2022, 2022.
- [22] *Api blueprint*, Accessed: Jun 3, 2022, 2021.
- [23] M. U. Hadi *et al.*, *Large language models: A comprehensive survey of its applications, challenges, limitations, and future prospects*, 2023.
- [24] OpenAI, *Gpt-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL].
- [25] T. Brown *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

- [26] D. Baidoo-Anu and L. O. Ansah, “Education in the era of generative artificial intelligence (ai): Understanding the potential benefits of chatgpt in promoting teaching and learning,” *Journal of AI*, vol. 7, no. 1, pp. 52–62, 2023.
- [27] P. A. C. Debby R. E. Cotton and J. R. Shipway, “Chatting and cheating: Ensuring academic integrity in the era of chatgpt,” *Innovations in Education and Teaching International*, vol. 61, no. 2, pp. 228–239, 2024. eprint: <https://doi.org/10.1080/14703297.2023.2190148>.
- [28] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, “Improving language understanding by generative pre-training,” 2018.
- [29] B. Min *et al.*, “Recent advances in natural language processing via large pre-trained language models: A survey,” *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–40, 2023.
- [30] B. Jacob *et al.*, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.
- [31] T. Dettmers and L. Zettlemoyer, “The case for 4-bit precision: K-bit inference scaling laws,” in *International Conference on Machine Learning*, PMLR, 2023, pp. 7750–7774.
- [32] L. Li, Q. Li, B. Zhang, and X. Chu, “Norm tweaking: High-performance low-bit quantization of large language models,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, 2024, pp. 18 536–18 544.
- [33] Z. Ali, H. Darwis, L. B. Ilmawan, S. R. Jabir, A. R. Manga, *et al.*, “Memory efficient with parameter efficient fine-tuning for code generation using quantization,” in *2024 18th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, IEEE, 2024, pp. 1–6.
- [34] Z. Liu *et al.*, “Llm-qat: Data-free quantization aware training for large language models,” *arXiv preprint arXiv:2305.17888*, 2023.
- [35] E. J. Hu *et al.*, “Lora: Low-rank adaptation of large language models,” *arXiv preprint arXiv:2106.09685*, 2021.
- [36] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “Qlora: Efficient fine-tuning of quantized llms,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

- [37] L. Reynolds and K. McDonell, “Prompt programming for large language models: Beyond the few-shot paradigm,” in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.
- [38] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [39] J. Wei *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.
- [40] J. Sun *et al.*, “Enhancing chain-of-thoughts prompting with iterative bootstrapping in large language models,” *arXiv preprint arXiv:2304.11657*, 2023.
- [41] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018, ISBN: 0262039249.
- [42] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.
- [43] L. Busoniu, R. Babuska, and B. De Schutter, “A comprehensive survey of multiagent reinforcement learning,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, 2008.
- [44] D. H. Wolpert and K. Tumer, “Optimal payoff functions for members of collectives,” *Advances in Complex Systems*, vol. 4, no. 02n03, pp. 265–279, 2001.
- [45] A. Arcuri, “Restful api automated test case generation with evomaster,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, Jan. 2019.
- [46] *Dredd*, Accessed: may 1, 2022, 2022.
- [47] Yelp, *Fuzz-lightyear*, 2023.
- [48] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, “Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’24, Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1383–1394.
- [49] National Institute of Standards and Technology, *Automated combinatorial testing for software*, Website, Available at <https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software>, accessed on April 30, 2025, 2023.

- [50] J. C. Alonso, A. Martin-Lopez, S. Segura, J. Garcia, and A. Ruiz-Cortes, “Arte: Automated generation of realistic test inputs for web apis,” *IEEE Transactions on Software Engineering*, vol. 49, no. 01, pp. 348–363, Jan. 2023.
- [51] C. Bizer *et al.*, “Dbpedia-a crystallization point for the web of data,” *Journal of web semantics*, vol. 7, no. 3, pp. 154–165, 2009.
- [52] H. Cao, J.-R. Falleri, and X. Blanc, “Automated generation of rest api specification from plain html documentation,” in *Service-Oriented Computing*, M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol, Eds., New York, NY, USA: Springer International Publishing, 2017, pp. 453–461, ISBN: 978-3-319-69035-3.
- [53] J. Yang, E. Wittern, A. T. T. Ying, J. Dolby, and L. Tan, “Towards extracting web api specifications from documentation,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR ’18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 454–464, ISBN: 9781450357166.
- [54] M. Badri, L. Badri, and M. Naha, “A use case driven testing process: Towards a formal approach based on uml collaboration diagrams,” in *Formal Approaches to Software Testing*, A. Petrenko and A. Ulrich, Eds., Berlin, Germany: Springer Berlin Heidelberg, 2004, pp. 223–235, ISBN: 978-3-540-24617-6.
- [55] J. Ryser and M. Glinz, *A scenario-based approach to validating and testing software systems using statecharts*, 1999.
- [56] A. Blasi, A. Gorla, M. D. Ernst, and M. Pezze, “Call me maybe: Using nlp to automatically generate unit test cases respecting temporal constraints,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22, Rochester, MI, USA: Association for Computing Machinery, 2022, ISBN: 9781450394758.
- [57] A. Blasi *et al.*, “Translating code comments to procedure specifications,” in *Proceedings of the 2018 International Symposium on Software Testing and Analysis (ISSTA 2018)*, ser. ISSTA ’18, Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 242–253, ISBN: 9781450356992.
- [58] C. Wang, F. Pastore, A. Goknil, and L. C. Briand, “Automatic generation of acceptance test cases from use case specifications: An nlp-based approach,” *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 585–616, 2022.
- [59] C. Wang, F. Pastore, and L. Briand, “Automated generation of constraints from use case specifications to support system testing,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, Piscataway, NJ, USA: IEEE, 2018, pp. 23–33.

- [60] M. Motwani and Y. Brun, “Automatically generating precise oracles from structured natural language specifications,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 188–199.
- [61] F. Barros, L. Neves, E. Hori, and D. Torres, “The ucscnl: A controlled natural language for use case specifications,” in *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, Miami Beach, FL, USA: KSI Research Inc, Jan. 2011, pp. 250–253.
- [62] M. Badri, L. Badri, and M. Naha, “A use case driven testing process: Towards a formal approach based on uml collaboration diagrams,” in *Formal Approaches to Software Testing*, A. Petrenko and A. Ulrich, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 223–235, ISBN: 978-3-540-24617-6.
- [63] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, “Requirements by contracts allow automated system testing,” in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 85–96.
- [64] J. Ryser and M. Glinz, *A scenario-based approach to validating and testing software systems using statecharts*, 1999.
- [65] L. Tahat, B. Vaysburg, B. Korel, and A. Bader, “Requirement-based automated black-box test generation,” in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, 2001, pp. 489–495.
- [66] T. Yue, S. Ali, and M. Zhang, “Rtcm: A natural language based, automated, and practical test case generation framework,” in *Proceedings of the 2015 international symposium on software testing and analysis*, 2015, pp. 397–408.
- [67] Y. Zheng *et al.*, “Automatic web testing using curiosity-driven reinforcement learning,” in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE ’21, Madrid, Spain: IEEE Press, 2021, pp. 423–435, ISBN: 9781450390859.
- [68] Y. Koroglu *et al.*, “QBE: QLearning-Based Exploration of Android Applications,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, 2018, pp. 105–115.
- [69] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, “Reinforcement learning based curiosity-driven testing of android applications,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSSTA 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 153–164, ISBN: 9781450380089.

- [70] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, “Reinforcement learning for android gui testing,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 2–8, ISBN: 9781450360531.
- [71] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, “Autoblacktest: Automatic black-box testing of interactive applications,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 81–90.
- [72] T. A. T. Vuong and S. Takada, “A reinforcement learning based approach to automated testing of android applications,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 31–37, ISBN: 9781450360531.
- [73] Y. K orođlu and A. Sen, “Functional test generation from ui test scenarios using reinforcement learning for android applications,” *Software Testing, Verification and Reliability*, vol. 31, Oct. 2020.
- [74] K. Basu *et al.*, “Nestful: A benchmark for evaluating llms on nested sequences of api calls,” *arXiv preprint arXiv:2409.03797*, 2024.
- [75] A. Decrop, G. Perrouin, M. Papadakis, X. Devroey, and P.-Y. Schobbens, “You can rest now: Automated specification inference and black-box testing of restful apis with large language models,” *arXiv preprint arXiv:2402.05102*, 2024.
- [76] Y. Song *et al.*, “Restgpt: Connecting large language models with real-world restful apis,” *arXiv preprint arXiv:2306.06624*, 2023.
- [77] S. Newman, *Building Microservices*, 1st. O’Reilly Media, 2015, ISBN: 1491950358.
- [78] A. Arcuri, “Restful api automated test case generation,” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Prague, Czech Republic: IEEE, 2017, pp. 9–20.
- [79] P. Godefroid, B.-Y. Huang, and M. Polishchuk, “Intelligent rest api data fuzzing,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 725–736.
- [80] A. Martin-Lopez, S. Segura, and A. Ruiz-Cort es, “Restest: Black-box constraint-based testing of restful web apis,” in *International Conference on Service-Oriented Computing*, Springer, 2020, pp. 459–475.

- [81] E. Viglianisi, M. Dallago, and M. Ceccato, “Resttestgen: Automated black-box testing of restful apis,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, 2020, pp. 142–152.
- [82] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, “Automated black-box testing of nominal and error scenarios in restful apis,” *Software Testing, Verification and Reliability*, e1808, 2022.
- [83] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Checking security properties of cloud service rest apis,” in *13th International Conference on Software Testing, Validation and Verification (ICST)*, Porto, Portugal: IEEE, 2020, pp. 387–397.
- [84] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, *Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations*, 2020. arXiv: 2005.11498 [cs.SE].
- [85] N. Laranjeiro, J. Agnelo, and J. Bernardino, “A black box tool for robustness testing of rest services,” *IEEE Access*, pp. 24 738–24 754, 2021.
- [86] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, “Metamorphic testing of restful web apis,” *IEEE Transactions on Software Engineering (TSE)*, pp. 1083–1099, 2017.
- [87] H. Ed-Douibi, J. L. C. Izquierdo, and J. Cabot, “Automatic generation of test cases for rest apis: A specification-based approach,” in *22nd International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2018, pp. 181–190.
- [88] D. Stallenberg, M. Olsthoorn, and A. Panichella, “Improving test case generation for rest apis through hierarchical clustering,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 117–128.
- [89] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, “Empirical comparison of black-box test case generation tools for restful apis,” in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Luxembourg: IEEE, 2021, pp. 226–236.
- [90] Z. Hatfield-Dodds and D. Dygalo, “Deriving semantics-aware fuzzers from web api schemas,” *arXiv preprint arXiv:2112.10328*, 2021.
- [91] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?” In *Proceedings of the 32nd International Symposium on Software Reliability Engineering*, to appear, 2021.
- [92] *Apifuzzer*, 2023.

- [93] A. Arcuri, “Automated black-and white-box testing of restful apis with evomaster,” *IEEE Software*, vol. 38, no. 3, pp. 72–78, 2020.
- [94] SE@GT, *Experiment infrastructure, data, and results for restgpt (github)*, <https://github.com/selab-gatech/RESTGPT>, 2024.
- [95] *Evomaster*, Accessed: Jun 3, 2022, 2022.
- [96] *Restler*, Accessed: Jun 3, 2022, 2022.
- [97] *Restest*, Accessed: Jun 3, 2022, 2022.
- [98] *Schemathesis*, Accessed: Jun 1, 2022, 2022.
- [99] *Bboxrt*, Accessed: Jun 3, 2022, 2022.
- [100] *Cats*, Accessed: Jun 3, 2022, 2022.
- [101] *Gotswag*, Accessed: Jun 3, 2022, 2018.
- [102] A. Arcuri, “Evomaster: Evolutionary multi-context automated system test generation,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, Västerås, Sweden: IEEE, 2018, pp. 394–397.
- [103] A. Arcuri, “Many independent objective (MIO) algorithm for test suite generation,” *CoRR*, vol. abs/1901.01541, pp. 3–17, 2019. arXiv: 1901.01541.
- [104] A. Arcuri and J. P. Galeotti, “SQL data generation to enhance search-based system testing,” in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO July 13-17, 2019*, A. Auger and T. Stützle, Eds., Prague, Czech Republic: ACM, 2019, pp. 1390–1398.
- [105] M. Zhang, B. Marculescu, and A. Arcuri, “Resource and dependency based test case generation for restful web services,” *Empirical Software Engineering*, pp. 1–61, 2021.
- [106] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “A catalogue of inter-parameter dependencies in restful web apis,” in *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings*, Toulouse, France: Springer-Verlag, 2019, pp. 399–414, ISBN: 978-3-030-33701-8.
- [107] A. Martin-Lopez, S. Segura, C. Muller, and A. Ruiz-Cortes, “Specification and automated analysis of inter-parameter dependencies in web apis,” *IEEE Transactions on Services Computing*, pp. 1–1, 2021.

- [108] *Idlreasoner*, Accessed: May 1, 2022, 2022.
- [109] *Hypothesis*, Accessed: Jun 3, 2022, 2022.
- [110] *Gavel*, Accessed: Jun 3, 2022, 2022.
- [111] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to combinatorial testing*. CRC press, 2013.
- [112] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, “Comparing operating systems using robustness benchmarks,” in *Proceedings of 16th IEEE Symposium on Reliable Distributed Systems (SRDS)*, IEEE, 1997, pp. 72–79.
- [113] *Companion page with experiment infrastructure, data, and results for rest-go project*, Accessed: April 30, 2024, 2022.
- [114] *Jacoco*, Accessed: Jun 3, 2022, 2021.
- [115] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [116] M. Saad *et al.*, “Exploring the attack surface of blockchain: A systematic overview,” *arXiv preprint arXiv:1904.03487*, 2019.
- [117] D. Freedman, R. Pisani, and R. Purves, *Statistics (international student edition)*. WW Norton & Company, 2007.
- [118] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [119] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.
- [120] D. Vosta, *Evaluation of the t-wise approach for testing rest apis*, 2020.
- [121] C. Manning and H. Schutze, *Foundations of statistical natural language processing*. MIT press, 1999.
- [122] J. J. Webster and C. Kit, “Tokenization as the initial phase in nlp,” in *COLING 1992 Volume 4: The 14th International Conference on Computational Linguistics*, 1992.
- [123] A. Voutilainen, “Part-of-speech tagging,” *The Oxford handbook of computational linguistics*, pp. 219–232, 2003.

- [124] S. Kübler, R. McDonald, and J. Nivre, “Dependency parsing,” *Synthesis lectures on human language technologies*, pp. 1–127, 2009.
- [125] nltk.or, *Nltk wordnet*, <https://www.nltk.org/howto/wordnet.html>, 2023.
- [126] M. I. Jordan and T. M. Mitchell, “Machine learning: Trends, perspectives, and prospects,” *Science*, pp. 255–260, 2015.
- [127] X. Zhu and A. B. Goldberg, “Introduction to semi-supervised learning,” *Synthesis lectures on artificial intelligence and machine learning*, pp. 1–130, 2009.
- [128] LanguageTool, *Languagetool rest api*, <https://languagetool.org/proofreading-api>, 2023.
- [129] B. Marculescu, M. Zhang, and A. Arcuri, “On the faults found in REST APIs by automated test generation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, Mar. 2022.
- [130] H. Wu, L. Xu, X. Niu, and C. Nie, “Combinatorial testing of restful apis,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22, New York, NY, USA: Association for Computing Machinery, 2022, pp. 426–437, ISBN: 9781450392211.
- [131] Myeongsoo Kim and Davide Corradini, *Experiment infrastructure and data for nlp2rest*, <https://github.com/codingsoo/nlp2rest>, 2023.
- [132] S. Software, *Openapi extensions*, <https://swagger.io/docs/specification/openapi-extensions/>, 2023.
- [133] A. Martin-Lopez, S. Segura, C. Müller, and A. Ruiz-Cortés, “Specification and automated analysis of inter-parameter dependencies in web apis,” *IEEE Transactions on Services Computing*, vol. 15, no. 4, pp. 2342–2355, 2022.
- [134] Y. Goldberg and O. Levy, *Word2vec explained: Deriving mikolov et al.’s negative-sampling word-embedding method*, 2014. arXiv: 1402.3722 [cs.CL].
- [135] T. L. Foundation, *Openapi specification*, <https://spec.openapis.org/oas/v3.1.0>, 2022.
- [136] D. Klein and C. D. Manning, “Accurate unlexicalized parsing,” in *Proceedings of the 41st annual meeting of the association for computational linguistics*, Edinburgh, Scotland: Association for Computational Linguistics, 2003, pp. 423–430.

- [137] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, *Fasttext.zip: Compressing text classification models*, 2016. arXiv: 1612.03651 [cs.CL].
- [138] D. Jurafsky and J. H. Martin, *Speech and language processing: Constituency parsing*, <https://web.stanford.edu/~jurafsky/slp3/13.pdf>, 2021.
- [139] N. Hardeniya, J. Perkins, D. Chopra, N. Joshi, and I. Mathur, *Natural language processing: python and NLTK*. Birmingham, UK: Packt Publishing Ltd, 2016.
- [140] A. Martin-Lopez, S. Segura, and A. 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*, ser. ESEC/FSE ’22, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 408–420, ISBN: 9781450394130.
- [141] M. Kim, Q. Xin, S. Sinha, and A. Orso, “Automated test generation for REST apis: No time to rest yet,” in *ISSTA ’22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds., New York, NY, USA: ACM, 2022, pp. 289–301.
- [142] Y. Liu *et al.*, “Morest: Model-based restful api testing with execution feedback,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1406–1417, ISBN: 9781450392211.
- [143] N. Laranjeiro, J. Agnelo, and J. Bernardino, “A black box tool for robustness testing of rest services,” *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021.
- [144] M. Kim *et al.*, “Enhancing REST API Testing with NLP Techniques,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, 2023, pp. 1232–1243.
- [145] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.
- [146] E. Perez, D. Kiela, and K. Cho, “True few-shot learning with language models,” *Advances in neural information processing systems*, vol. 34, pp. 11 054–11 070, 2021.
- [147] *Openai cookbook*, <https://github.com/openai/openai-cookbook>, 2023.
- [148] *Promptbase*, <https://promptbase.com/>, 2023.

- [149] S. Software, *Openapi data model*, <https://swagger.io/docs/specification/data-models/keywords/>, 2023.
- [150] M. Kim, *Experiment infrastructure and data for arat-rl*, 2023.
- [151] *Q learning practical guide*, 2023.
- [152] N. Kumar, *Reinforcement learning guide with python example*, 2023.
- [153] A. Masadeh, Z. Wang, and A. E. Kamal, “Reinforcement learning exploration algorithms for energy harvesting communications systems,” in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6.
- [154] *Python random library*, 2023.
- [155] *Python datetime library*, 2023.
- [156] *Rstr library*, 2023.
- [157] *Difflib sequencematcher (gestalt pattern matching)*, 2023.
- [158] P. E. Black, *Ratcliff/obershelp pattern recognition*, 2021.
- [159] *Mitmproxy*, 2023.
- [160] M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, *Leveraging large language models to improve rest api testing*, 2023. arXiv: 2312.00894 [cs.SE].
- [161] Meta, *Meta-llama-3-8b*, 2024.
- [162] APIs.guru, *Apis-guru*, 2023.
- [163] A. Golmohammadi, M. Zhang, and A. Arcuri, “Testing restful apis: A survey,” *ACM Trans. Softw. Eng. Methodol.*, Aug. 2023.
- [164] M. Kim, *Experiment infrastructure, data, and results for llamaresttest*, <https://github.com/codingsoo/LlamaRestTest>, 2023.
- [165] H. Discussion, *Requirement of llama2-7b model*, 2024.
- [166] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, *Gptq: Accurate post-training quantization for generative pre-trained transformers*, 2023. arXiv: 2210.17323 [cs.LG].

- [167] Z. Yao *et al.*, *Hawqv3: Dyadic neural network quantization*, 2021. arXiv: 2011.10680 [cs.CV].
- [168] S. Shen *et al.*, “Q-bert: Hessian based ultra low precision quantization of bert,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 8815–8821, Apr. 2020.
- [169] Anonymous, *Experiment infrastructure, data, and results for llamaresttest*, <https://anonymous.4open.science/r/LlamaRestTest-66B7/>, 2023.
- [170] I. Vieira, W. Allred, S. Lankford, S. Castilho, and A. Way, *How much data is enough data? fine-tuning large language models for in-house translation: Performance evaluation across multiple dataset sizes*, 2024. arXiv: 2409.03454 [cs.CL].
- [171] Meta, *Llama3 fine-tuning guide*, 2024.
- [172] J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. Smith, *Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping*, 2020. arXiv: 2002.06305 [cs.CL].
- [173] GIScience, *Ohsome api github*, 2025.
- [174] GIScience, *Ohsome api online*, 2025.
- [175] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, “Llm.int8(): 8-bit matrix multiplication for transformers at scale,” in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS ’22, New Orleans, LA, USA: Curran Associates Inc., 2022, ISBN: 9781713871088.
- [176] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, “A survey of quantization methods for efficient neural network inference,” in *Low-power computer vision*, Chapman and Hall/CRC, 2022, pp. 291–326.
- [177] M. Kim, T. Stennett, S. Sinha, and A. Orso, *A multi-agent approach for rest api testing with semantic graphs and llm-driven inputs*, 2025. arXiv: 2411.07098 [cs.SE].
- [178] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Pearson, 2010.
- [179] M. Wooldridge, *An introduction to multiagent systems*. John Wiley & Sons, 2009.

- [180] K. Zhang, Z. Yang, and T. Başar, “Multi-agent reinforcement learning: A selective overview of theories and algorithms,” *Handbook of reinforcement learning and control*, pp. 321–384, 2021.
- [181] P. Sunehag *et al.*, “Value-decomposition networks for cooperative multi-agent learning,” *arXiv preprint arXiv:1706.05296*, 2017. arXiv: 1706.05296 [cs.LG].
- [182] *Error report for spotify*, 2024.
- [183] Anonymous, *Error report for ohsome*, 2024.
- [184] SE@GT, *Experiment infrastructure, data, and results for autoresttest*, <https://github.com/selab-gatech/AutoRestTest/>, 2024.
- [185] V. Kumar and M. Webster, “Importance sampling based exploration in q learning,” *arXiv preprint arXiv:2107.00602*, 2021. arXiv: 2107.00602 [cs.LG].
- [186] gibberblot, *Gibberblot’s blog post*, <https://gibberblot.github.io/rl-notes/single-agent/temporal-difference-learning.html>, 2024.
- [187] M. Kim, S. Pande, and A. Orso, “Improving program debloating with 1-du chain minimality,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion ’24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 384–385.
- [188] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.
- [189] N. Rekabsaz, M. Lupu, and A. Hanbury, “Exploration of a threshold for similarity based on uncertainty in word embedding,” in *European Conference on Information Retrieval*, Springer, 2017, pp. 396–409.
- [190] K. Komiya, Y. Abe, H. Morita, and Y. Kotani, “Question answering system using q & a site corpus query expansion and answer candidate evaluation,” *SpringerPlus*, vol. 2, pp. 1–11, 2013.
- [191] J. Yao, C. Yuan, X. Li, Y. Wang, and Y. Su, “Beyond top-k: Knowledge reasoning for multi-answer temporal questions based on revalidation framework,” *PeerJ Computer Science*, vol. 9, e1725, 2023.
- [192] M. Li, X. Shen, Y. Sun, W. Zhang, J. Nan, D. Gao, *et al.*, “Using semantic text similarity calculation for question matching in a rheumatoid arthritis question-answering system,” *Quantitative Imaging in Medicine and Surgery*, vol. 13, no. 4, p. 2183, 2023.

- [193] R. Huang, M. Motwani, I. Martinez, and A. Orso, “Generating rest api specifications through static analysis,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, Lisbon, Portugal: Association for Computing Machinery, 2024.