FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automated and Context-Aware API Contract Validation

Henrique Gonçalves Graveto Futuro da Silva

Mestrado em Engenharia Informática e Computação

Supervisor: Bruno Lima

Second Supervisor: João Pedro Dias

July 7, 2026

© Henrique Gonçalves Graveto Futuro da Silva, 2026

Automated and Context-Aware API Contract Validation

Henrique Gonçalves Graveto Futuro da Silva

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. João Pascoal Faria
Examiner: Prof. Ângelo Martins
Member: Prof. Bruno Lima

July 7, 2026

Resumo

Teste de software é fundamental para garantir que as aplicações funcionam conforme o esperado. À medida que o software aumentou em complexidade e as aplicações se tornaram cada vez mais interligadas, tornou-se cada vez mais importante garantir que as especificações das Interfaces de Programação de Aplicações (API) descrevam corretamente o seu uso pretendido, bem como garantir a sua consistência e conformidade com as diretrizes organizacionais. Consequentemente, foram desenvolvidas várias abordagens para gerar casos de teste e dados a partir das especificações existentes, assim como mecanismos para definir restrições nas especificações. Adicionalmente, *API linters* foram desenvolvidos para validar automaticamente especificações contra um conjunto predefinido de regras, assegurando a sua correção e consistência. No entanto, uma limitação destas abordagens reside no facto de não compreenderem o contexto da especificação nem das diretrizes organizacionais, restringindo as suas capacidades de deteção a problemas sintáticos e estruturais bem definidos.

Com o rápido crescimento da Inteligência Artificial (IA) e dos Modelos de Linguagem de Larga Escala (LLM) nos últimos anos, surgiram soluções que integram IA, tirando partido da capacidade destes modelos para interpretar linguagem natural e raciocinar sobre o seu significado. Apesar disso, atualmente não existe nenhuma solução automatizada e escalável capaz de integrar diretrizes organizacionais no processo de validação e raciocinar sobre o contexto em que estas devem ser aplicadas.

Esta dissertação apresenta o NysusAI, uma ferramenta automatizada para a validação contextual de contratos de APIs desenvolvida em colaboração com a Kuehne+Nagel, com o objetivo de abordar as limitações das ferramentas de validação existentes. O NysusAI utiliza LLMs e técnicas avançadas de engenharia de *prompts* para validar automaticamente especificações OpenAPI contra diretrizes organizacionais. Para garantir escalabilidade, cada pedido enviado à LLM é composto por uma porção da especificação e as diretrizes necessárias para a sua validação, em vez de processar a especificação completa e todas as diretrizes simultaneamente. Adicionalmente, o NysusAI é capaz de detetar inconsistências internas da própria especificação.

Para validar o NysusAI, foram realizadas duas experiências. A primeira avaliou a fiabilidade dos resultados num conjunto de 50 especificações OpenAPI de empresas reais, obtendo uma taxa de falsas descobertas de 18.6%, com as violações das diretrizes a atingir 8.5% e a deteção de inconsistências a atingir 34.5%. Esta experiência também demonstrou que o NysusAI detetou 133 verdadeiros positivos em violações das diretrizes em 16 diretrizes dependentes de contexto e 133 inconsistências dependentes de contexto em sete categorias. A segunda experiência avaliou o impacto do NysusAI na Experiência do Desenvolvedor (DX) através de um estudo com 10 participantes, utilizando o Modelo de Aceitação de Tecnologia (TAM). Os resultados mostraram médias compostas de 4.33, 4.30 e 3.95 para a Perceção de Facilidade de Uso (PEOU), Perceção de Utilidade (PU) e Intenção de Uso (ITU), respetivamente, todas acima do valor neutro de 3.0, indicando que o NysusAI é percecionado como fácil de usar, útil, e que vale a pena adotar.

Todos os resultados obtidos suportam a hipótese de que uma ferramenta automatizada para a

validação contextual de contratos de APIs é capaz de detetar violações de design e inconsistências que ferramentas tradicionais de análise estática não conseguem detetar, contribuindo simultaneamente para a melhoria da DX.

Palavras-chave: Validação Automatizada de Contratos de API • Validação Contextual • Modelos de Linguagem de Larga Escala • Engenharia de Prompts • Conformidade e Normas de APIs • Especificação OpenAPI

Abstract

Software testing is crucial to ensure that applications work as intended. As software grew in complexity and applications became increasingly interconnected, it became extremely important to make sure that Application Programming Interface (API) specifications correctly described their intended use, as well as ensuring they are consistent and compliant with organizational guidelines. As a result, developers started creating approaches that generated test cases and input data from existing specifications, as well as ways to define constraints in the specification. Additionally, API linters were developed to automatically validate specifications against a predefined ruleset of constraints, ensuring correctness and consistency. However, one limitation of these tools is the fact that they do not grasp the context of the specification or the organizational guidelines, restricting their detection capabilities to well-defined syntactic and structural issues.

With the rapid growth of Artificial Intelligence (AI) and Large Language Model (LLM)s in recent years, developers started leveraging their capacity to read natural language and reason about its intent, integrating AI into validation tools. Despite this, there is currently no automated and scalable solution capable of integrating organizational guidelines into the validation process while reasoning about their context.

This dissertation presents NysusAI, an automated and context-aware API contract validation tool developed in collaboration with Kuehne+Nagel, addressing the gap left by existing validation tools. NysusAI leverages LLMs and advanced prompt engineering techniques to automatically validate OpenAPI Specifications against organizational guidelines. To ensure scalability, each call to the LLM is composed of a targeted portion of the specification alongside only the relevant guidelines needed for its validation, rather than processing the entire specification and all guidelines at once. Additionally, NysusAI detects internal inconsistencies within the specification itself.

To validate NysusAI, two experiments were conducted, where the first assessed the reliability of NysusAI's output across a dataset of 50 real-world OpenAPI Specifications, achieving an overall false discovery rate of 18.6%, with guideline violations achieving 8.5% and inconsistency detection achieving 34.5%. This experiment also demonstrated that NysusAI detected 133 true positive guideline violations across 16 context-dependent guidelines and 133 context-dependent inconsistencies across seven categories. The second experiment evaluated the NysusAI's impact on Developer Experience (DX) through a user study with 10 participants, using the Technology Acceptance Model (TAM) methodology. The results showed composite means of 4.33, 4.30, and 3.95 for Perceived Ease of Use (PEOU), Perceived Usefulness (PU), and Intention to Use (ITU) respectively, all above the neutral value of 3.0, indicating that NysusAI is perceived as easy to use, useful, and worth adopting.

All of these results support the hypothesis that an automated and context-aware API contract validation tool is capable of detecting design violations and inconsistencies that traditional static analysis tools cannot identify, while improving DX.

Keywords: Automated API Contract Validation • Context-Aware Validation • Large Language Models • Prompt Engineering • API Governance • OpenAPI Specification

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (https://sdgs.un.org/goals) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice. This dissertation presents NysusAI, an automated and context-aware Application Programming Interface (API) contract validation tool that contributes to the improvement of software quality and developer productivity. By reducing the manual effort required to validate API specifications and detecting issues that existing static analysis tools cannot identify, NysusAI supports more efficient and reliable software development practices.

The specific Sustainable Development Goals mentioned have the following names:

SDG 8 Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all.

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

SDG 17 Strengthen the means of implementation and revitalize the Global Partnership for Sustainable Development.

SDG	Target	Contribution	Performance Indicators and Metrics
8	8.2	NysusAI improves developer productivity by automating the API contract validation process, reducing the manual effort required and increasing the efficiency of software development teams.	Reduction in manual effort required for API validation; improvement in developer productivity as measured by Technology Acceptance Model (TAM) composite means of 4.33, 4.30, and 3.95 for Perceived Ease of Use (PEOU), Perceived Usefulness (PU), and Intention to Use (ITU) respectively.
	8.4	NysusAI reduces the manual effort required for API contract review, contributing to more efficient use of human resources in software development.	Reduction in manual effort required for API validation; number of API specification issues detected that would otherwise require manual identification.
9	9.1	NysusAI improves the quality and reliability of API specifications, which form the foundation of modern digital infrastructure, by detecting issues that existing static tools cannot identify.	Number of API specification issues detected and resolved; improvement in API specification quality.
	9.5	NysusAI fosters innovation in software engineering by introducing a context-aware and automated approach to API contract validation, going beyond existing static analysis tools and improving the quality and reliability of digital infrastructure.	Number of API specification issues detected that static tools cannot identify;
17	17.6	This dissertation results from an academia-industry collaboration between Faculdade de Engenharia da Universidade do Porto (FEUP) and Kuehne+Nagel, promoting knowledge sharing and technology transfer between research institutions and industry.	Establishment of a working prototype validated with real-world industry scenarios provided by Kuehne+Nagel.
	17.16	The collaboration between FEUP and Kuehne+Nagel strengthens multi-stakeholder partnerships, combining academic research with industrial expertise to develop a practical and industry-relevant validation tool.	Number of real-world API specifications used for validation; number of guideline violations and inconsistencies detected across industry-provided specifications.

Acknowledgements

I would like to thank both my supervisors, Bruno Lima and João Pedro Dias, for accompanying me throughout this dissertation. Without your orientation and feedback, none of this work would have been possible.

I would also like to thank Kuehne+Nagel for this opportunity and for welcoming me with open arms. I would like to give a special thanks to João Pereira for the guidance, feedback, and knowledge shared throughout the development of this dissertation.

I would like to thank all of my colleagues who accompanied me throughout this long journey, for all the laughter and good moments we shared together.

Lastly, I would like to thank all of my friends and family for all of the support and strength they gave me. Without it, I would not have been able to complete this dissertation.

To all, thank you.

Henrique Silva

Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.

Henrique Gonçalves Graveto Futuro da Silva

"We don't make mistakes, we have happy accidents."

Robert "Bob Ross" Norman

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Problem . . . 2
- 1.4 Hypothesis . . . 3
- 1.5 Research Questions . . . 3
- 1.6 Dissertation Structure . . . 4

2 Background & State of the art **5**
- 2.1 Background . . . 5
  - 2.1.1 Web APIs . . . 5
  - 2.1.2 Synchronous APIs . . . 9
- 2.2 State of the Art . . . 11
  - 2.2.1 Research Domain . . . 11
  - 2.2.2 Identification Phase . . . 12
  - 2.2.3 Refinement Process . . . 13
  - 2.2.4 Screening Phase . . . 14
  - 2.2.5 Eligibility Phase . . . 15
  - 2.2.6 Results and Discussion . . . 19
  - 2.2.7 Constraint-Based API Testing . . . 20
  - 2.2.8 Automated Test Case and Input Generation for REST APIs . . . 25
  - 2.2.9 Related Work . . . 28
  - 2.2.10 Analysis of State of the Art . . . 30
- 2.3 Summary . . . 31

3 NysusAI **32**
- 3.1 *Desiderata* . . . 32
- 3.2 Tool Context . . . 33
  - 3.2.1 Zalando RESTful API and Event Guidelines . . . 33
  - 3.2.2 OpenAPI Specifications Dataset . . . 33
- 3.3 Architecture . . . 34
  - 3.3.1 Scraper Module . . . 35
  - 3.3.2 Database . . . 38
  - 3.3.3 LLM Module . . . 38
  - 3.3.4 Validator Module . . . 43
  - 3.3.5 Report Module . . . 47
  - 3.3.6 NysusAI Interface . . . 48
- 3.4 Summary . . . 49

4 Validation **51**
4.1 Output Reliability Evaluation . . . 51
4.1.1 Experimental Setup . . . 51
4.1.2 Experimental Results . . . 52
4.1.3 Experimental Conclusions . . . 58
4.2 User Study . . . 59
4.2.1 Experimental Setup . . . 59
4.2.2 User Profiling . . . 60
4.2.3 TAM Evaluation . . . 62
4.2.4 Experimental Conclusions . . . 64
4.3 Research Statement Revisited . . . 64
4.4 Threats to Validity . . . 67
4.5 Limitations . . . 67
4.6 Summary . . . 68

5 Conclusion and Future Work **70**
5.1 Conclusion . . . 70
5.2 Future Work . . . 71

References **72**

A Search Strings **82**

B OpenAPI Specifications Dataset **84**

C Prompts **88**
C.1 Deduplication Prompt . . . 88
C.2 Guidelines Validation Prompt . . . 89
C.3 Inconsistencies Detection Prompt . . . 91

D Token Usage Analysis **94**

E Example Validation Report **97**

F User Survey **102**

G TAM Survey Results **108**

List of Figures

2.1 API lifecycle overview. . . . 6
2.2 REST API overview. . . . 8
2.3 Synchronous API call. . . . 10
2.4 PRISMA flowchart. . . . 12
2.5 Papers statistics per year. . . . 19
2.6 OAS extended with IDL dependencies. . . . 21
2.7 BNF used for representing parameter constraints. . . . 22
2.8 RESTInfer overview. . . . 22
2.9 NLPtoREST overview. . . . 23
2.10 Constituency parse tree of the sentence “For example, en-US”. . . . 23
2.11 Overview of RESTest. . . . 25
2.12 ODG representing a dependency between two endpoints. . . . 26
2.13 Example of a Spectral Rule. . . . 29

3.1 Example of a Zalando guideline. . . . 33
3.2 Architecture of NysusAI. . . . 35
3.3 Scraper Module data flow diagram. . . . 36
3.4 Transformation of a Zalando guideline code snippet into structured output. . . . 38
3.5 Extraction of a table from a Zalando guideline. . . . 39
3.6 *LLM Module* data flow diagram. . . . 40
3.7 GitHub Copilot maximum prompt tokens for GPT-4.1. . . . 40
3.8 *Validator Module* data flow diagram. . . . 44
3.9 Example of a low-confidence false positive issue flagged by the LLM. . . . 46
3.10 *Report Module* data flow diagram. . . . 47
3.11 Example of issues grouped under the *$.components.schemas* sub-path. . . . 48
3.12 NysusAI GUI. . . . 48

4.1 Overall issues detection results per company. . . . 52
4.2 Guidelines violations detection results per company. . . . 53
4.3 Distribution of true positive guideline violations detectable by static tools. . . . 54
4.4 Distribution of true positive guideline violations requiring context-aware tools. . . . 55
4.5 Distribution of false positive guideline violations by guideline. . . . 56
4.6 Inconsistency detection results per company. . . . 57
4.7 Distribution of true positive inconsistencies and context-dependent inconsistencies. 59
4.8 Background distribution of the 10 participants. . . . 60
4.9 Participants’ knowledge with OpenAPI Specifications. . . . 61
4.10 Participants’ familiarity with Zalando guidelines and validation experience. . . . 61

G.1 PEOU survey results. . . . 108

G.2 PU survey results. 109
G.3 ITU survey results. 110

List of Tables

2.1 Initial search string used on Scopus. . . . 13
2.2 Final QS1 search string used on Scopus. . . . 14
2.3 Results using the refined search string for each database. . . . 14
2.4 Inclusion and exclusion criteria used in the screening phase. . . . 15
2.5 Selected papers from 2024. . . . 16
2.6 Selected papers from 2023. . . . 17
2.7 Selected papers from 2022. . . . 17
2.8 Selected papers from 2021. . . . 18
2.9 Selected papers from 2020. . . . 18
2.10 Selected paper from 2019. . . . 18

3.1 Distribution of OpenAPI Specifications by company and domain. . . . 34
3.2 Comparison of the cloud-hosted database solutions considered. . . . 38
3.3 Models accessible with the Copilot Student plan. . . . 41
3.4 Average token usage ratio across API specifications by company. . . . 46

4.1 Perceived Ease of Use results. . . . 63
4.2 Perceived Usefulness results. . . . 64
4.3 Intention to Use results. . . . 65

A.1 QS1 refined search string used on each database. . . . 83

B.1 Adyen OpenAPI Specifications. . . . 84
B.2 Amadeus OpenAPI Specifications. . . . 84
B.3 Brex OpenAPI Specifications. . . . 85
B.4 Countrylayer OpenAPI Specifications. . . . 85
B.5 DHL OpenAPI Specifications. . . . 85
B.6 Youtube OpenAPI Specifications. . . . 85
B.7 Kong OpenAPI Specifications. . . . 86
B.8 Kuehne+Nagel OpenAPI Specifications. . . . 87

D.1 Token usage for the Adyen API specifications. . . . 94
D.2 Token usage for the Amadeus API specifications. . . . 94
D.3 Token usage for the Brex API specifications. . . . 95
D.4 Token usage for the Countrylayer API specifications. . . . 95
D.5 Token usage for the DHL API specifications. . . . 95
D.6 Token usage for the Youtube API specifications. . . . 95
D.7 Token usage for the Kong API specifications. . . . 96
D.8 Token usage for the Kuehne+Nagel API specifications. . . . 96

List of Acronyms

ABNF	Augmented Backus–Naur Form
AFIPS	American Federation of Information Processing Societies
AI	Artificial Intelligence
API	Application Programming Interface
BNF	Backus-Naur Form
CI/CD	Continuous Integration and Continuous Delivery
CLI	Command Line Interface
CNN	Convolutional Neural Network
CRUD	Create, Read, Update, Delete
CT	Combinatorial Testing
DOI	Digital Object Identifier
DRL	Deep Reinforcement Learning
DSL	Domain-Specific Language
DX	Developer Experience
FEUP	Faculdade de Engenharia da Universidade do Porto
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDL	Inter-parameter Dependency Language
IT	Information Technology
ITU	Intention to Use
ISBN	International Standard Book Number
JSON	JavaScript Object Notation
LLM	Large Language Model
MLP	Multilayer Perceptron
NER	Named-Entity Recognition
NLP	Natural Language Processing
OAD	OpenAPI Description
OAS	OpenAPI Specification
ODG	Operation Dependency Graph

PBT Property-based Testing
PEOU Perceived Ease of Use
PoS Part-of-Speech
PRISMA Preferred Reporting Items for Systematic reviews and Meta-Analyses
PU Perceived Usefulness
REST Representational State Transfer
RNN Recurrent Neural Network
RPC Remote Procedure Call
RPG RESTful-service Property Graph
SDG Sustainable Development Goal
SMTP Simple Mail Transfer Protocol
SOAP Simple Object Access Protocol
SUT System Under Test
TAM Technology Acceptance Model
TCP Transmission Control Protocol
TLS Transport Layer Security
TSL Test Specification Language
URI Uniform Resource Identifier
URL Uniform Resource Locator
XML Extensible Markup Language
YAML Yet Another Markup Language

Chapter 1

Introduction

In this chapter, an overall description of the research context and the objective of this thesis is presented. Section 1.1 introduces the background and key concepts. The motivation is presented in Section 1.2, highlighting the possible impact of the work in the industry. In Section 1.3 the problem is presented, defining the gaps between current solutions and the problem addressed in this thesis. Section 1.4 presents the hypothesis to address the identified problem. The research questions defined to guide the validation of the hypothesis are presented in Section 1.5. Section 1.6 presents the structure of this dissertation.

1.1 Context

Today, an Application Programming Interface (API) can be defined as a set of rules or protocols that enables software applications to communicate with each other to exchange data, features and functionality [28]. According to Postman[1], an average application uses between 26 and 50 APIs [54]. Given the central role APIs play in software applications, developers started to document them. One example is the creation of API specifications, a document that formalizes how APIs are shared. It can be seen as a machine-readable contract between the API provider and the consumer, describing the endpoints, request and response structures, and usage rules [4]. Over time, multiple standards have been created, such as the OpenAPI Specification (OAS)[2] and the AsyncAPI[3] Specification, ensuring consistency and reliability while allowing automation of tasks such as testing and validation [4]. To validate these specifications, developers created multiple approaches, a practice known as API contract validation. These approaches range from the creation of test cases and input data based on the specification, to more advanced constraint-based validation approaches that verify whether the API implementation conforms to the rules defined in the contract. API specifications can also be analyzed with API linters. These are tools that use rulesets or Artificial Intelligence (AI) to verify whether the specification follows a set of predefined constraints, ensuring data formats and data consistency, a practice known as API linting

[1] https://www.postman.com/
[2] https://www.openapis.org/
[3] https://www.asyncapi.com/en

[117]. All of these approaches can be automated and usually are integrated into Continuous Integration and Continuous Delivery (CI/CD) pipelines, allowing teams to detect and fix issues before deployment.

This dissertation results in a collaboration between Faculdade de Engenharia da Universidade do Porto (FEUP)[4] and Kuehne+Nagel[5]. Kuehne+Nagel is a global transport logistics company based in Switzerland. Operating all over the world, Kuehne+Nagel provides several type of services, such as transportation and fulfillment, digital services, value-added services and supply chain management. Thanks to this collaboration, the solution will be validated using real-world case scenarios. Throughout the remainder of this dissertation, the terms *API contract*, *API specification*, and *specification* are used interchangeably.

1.2 Motivation

The misuse of APIs can have significant problems in today's world, causing software bugs, crashes, and vulnerabilities [9]. Despite the availability of validation tools, currently there is no automated and scalable solution capable of automatically integrating organizational guidelines into de validation process while reasoning about their context to validate the API specification. The lack of such tool forces developers to manually review every API to ensure it follows internal standards, which is a time-consuming process that escalates with the size of the guidelines collection and the number of APIs to validate. This problem is particularly relevant in an industrial context, where numerous guidelines must be consistently enforced across multiple teams and projects. Multiple Postman's State of the API Reports state that a developer spends approximately 10 to 20 hours a week working with APIs, where about 15% of that time is used to debugging and manual testing [52, 53]. By solving this problem, the validation process could be improved, reducing the manual effort required and improving the overall developer experience.

1.3 Problem

Existing API contract validation techniques are either focused on generating test cases and input data to validate whether an API implementation is described correctly by its specification, or focused on creating new approaches to specify constraints in the specification, and test constraints defined in the specification against the implementation. Current API linters can be divided into two categories: static-based linters and context-aware linters. The former are limited to verifying well-defined issues, failing to grasp higher-level architectural or conceptual aspects, such as internal standards of a company. The latter uses AI to validate specifications, but are not scalable for the industrial scenarios as they processes the entire specification and all manually defined constraints in a single prompt. Additionally, despite API linting tools being partially automated, they still require an extensive amount of manual work, as the rulesets used to describe the constraints

[4] https://www.up.pt/feup/en/
[5] https://www.kuehne-nagel.com/

must be manually defined and continuously maintained as organizational guidelines evolve. The absence of a fully automated and scalable tool capable of integrating and reasoning about organizational guidelines represents a clear research gap in the field of API contract validation, which this dissertation aims to address.

1.4 Hypothesis

Based on the problems outlined in the preceding section, the following hypothesis was formulated:

An automated and context-aware API contract validation tool that provides contextualized feedback on architectural principles and semantic correctness is capable of detecting design violations and inconsistencies that traditional static analysis tools cannot identify, while improving developer experience.

An automated and context-aware API contract validation tool refers to a tool that analyzes API contracts without the need for manual intervention, and is capable of grasping the contract's context, as well as external context such as organizational guidelines.

Contextualized feedback refers to feedback that takes into account the context of the contract, as well as the organizational guidelines.

Architectural principles and semantic correctness refers to the design rules and conventions defined by the organizational guidelines that specify how the API contract should be structured, as well as the inconsistencies within the contract itself.

Design violations and inconsistencies refers to architectural, semantic, and structural errors in the API specification, such as violations of external organizational guidelines, as well as inconsistencies between different parts of the specification.

Traditional static analysis tools refers to tools that can only perform a structural analysis, such as ensuring the correct data formats are being used, such as Spectral[6].

Developer experience refers to the usefulness of the tool, as well as the effort required during the validation of API contracts.

1.5 Research Questions

To help validate the defined hypothesis, the following research questions will be addressed to help validate whether the hypothesis is true:

RQ1 What are the common issues with API specifications that are not addressed by existing automated and static analysis tools?

RQ2 How can precise and accurate feedback be inferred from organization guidelines and existing API contracts?

[6] https://stoplight.io/open-source/spectral

RQ3 Does an automated context-aware validation tool improve Developer Experience (DX)?

RQ4 Does an automated context-aware validation tool detect issues in API contracts that static analysis tools cannot identify?

1.6 Dissertation Structure

In addition to the introduction, this dissertation contains four more chapters.

Chapter 2 presents all the concepts needed to understand this dissertation such as Web APIs, Representational State Transfer (REST) APIs and Synchronous APIs. Additionally, a literature review on API contract validation is performed, along with a presentation and analysis of the state of the art and related work. Chapter 3 describes NysusAI in detail, covering the desired features, the system architecture, the modules that compose the tool, and the data used for its development. Chapter 4 presents all of the experiments performed to validate NysusAI, including their setup, results, analysis, and threats to validity. The limitations of NysusAI are also presented. Chapter 5 presents the conclusions of this dissertation, as well as the main contributes, and identifies opportunities for improvement for future work.

Chapter 2

Background & State of the art

This chapter presents the core concepts and essential background knowledge required for this dissertation, followed by a literature review using the Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA)[1] methodology on API contract validation. The approach of each relevant paper is also presented.

Section 2.1 introduces the key concepts needed to understand this dissertation. Section 2.2 presents the literature review and state-of-the-art approaches for API contract validation. A summary of the main findings of this chapter is presented in Section 2.3.

2.1 Background

This section introduces the key concepts necessary to understand this dissertation, covering Web APIs, their lifecycle, types and architectural styles. REST APIs and OpenAPI Specifications are presented with more detail since these formats are used in this dissertation.

2.1.1 Web APIs

The term API first appeared in the paper "Data structures and techniques for remote computer graphics", published by Cotton and Greatorex in the American Federation of Information Processing Societies (AFIPS)[2] conference, in 1968. Although at the time the term was used to describe the interaction of a system with the rest of the software program [21], in the following decades, as software systems grew in complexity, the definition of an API expanded from local systems to enabling communication across networks and platforms. This evolution led to the creation of Web APIs. A Web API is a type of API that uses protocols to manipulate external resources that expose an application's data and functionality over the internet [28]. To expose these APIs to the public, developers started creating API documentation. This documentation, which is manually created, requires a good understanding of the API, as well as reviewing and continuously updating it [55]. Its main objective is to have a human-readable guide on how to use and integrate the API, along

[1] https://www.prisma-statement.org/

[2] https://archives.lib.umn.edu/repositories/3/resources/41

with the ability to obtain tutorials and examples [127]. In addition to API documentation, API specifications, also known as API contracts, outline the various endpoints, methods, parameters, and data formats that developers can use to access and manipulate the features provided by an application [4].

2.1.1.1 API Lifecycle

The API lifecycle shows how an API is created, from its design to its deprecation. Figure 2.1 shows an overview of every stage that takes place in the lifecycle of an API. Each stage is explained below.

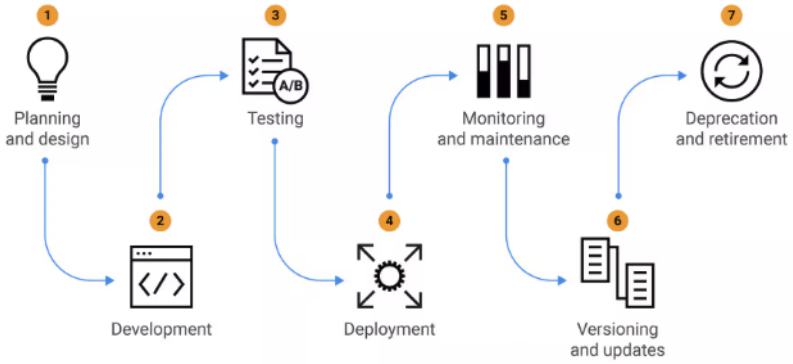


Figure 2.1: API lifecycle overview, illustrating the seven stages from planning and design to deprecation and retirement [5].

Planning and design The API's strategy, purpose, functionality, and architecture are established [5].

Development The development team starts to code and configure the API [5].

Testing The API is tested to identify and fix any vulnerability or bug in the software, and to make sure the API matches the specification [5].

Deployment After the testing phase is completed, the API is deployed to the production environment [5].

Monitoring and maintenance The performance, usage, and security are continuously monitored to make sure the API is working normally [5].

Versioning and updates New versions of the API are released to fix bugs and update the API [5].

Deprecation and retirement An API becomes inactive due to significant issues that may require a new version or because it has become obsolete [5].

2.1.1.2 Types of Web APIs

Depending on the scope of use, Web APIs can be labeled as different types. The main types are:

Open APIs Open APIs are public APIs, which have defined endpoints and request and response formats, and can be accessed with Hypertext Transfer Protocol (HTTP) requests [28].

Internal APIs Internal APIs are private APIs that are not available to external users [28].

Partner APIs Partner APIs are used to connect strategic business partners. This type of API is used by authorized external developers [28].

Composite APIs Composite APIs are used with the microservices architecture and allow developers to access several endpoints in a single call [28].

2.1.1.3 Architectural Styles of Web APIs

Depending on the architecture, Web APIs can be categorized differently. The main architectures are:

SOAP Simple Object Access Protocol (SOAP) is a lightweight XML-based messaging protocol specification that enables endpoints to receive and send data through several communication protocols, such as HTTP and Simple Mail Transfer Protocol (SMTP) [28].

RPC Remote Procedure Call (RPC) implements a logical client-to-server communication system designed for the support of network applications. It enables users to work with remote procedures as if the procedures were local. Transmitted data can be either in Extensible Markup Language (XML) (XML-RPC) or JavaScript Object Notation (JSON) (JSON-RPC) format. [28]

WebSocket WebSocket enables bidirectional communication between the server and client. This architecture is very appealing for real-time communication because it does not require a new connection to be established for each communication. Once a connection is established, it allows for continuous exchange [28].

REST REST defines a set of architecture principles. APIs that adhere to these principles are called REST APIs or RESTful APIs. REST APIs use HTTP requests to interact with resources. REST makes data available as resources, with each resource being identified by a unique Uniform Resource Identifier (URI) [28].

GraphQL GraphQL is a query language that specifies how clients should interact with APIs. It allows to make API requests with a query, instead of having to access complex endpoints with many parameters [28].

Since this dissertation focuses on REST APIs, this architecture will be explored in the following subsubsection.

2.1.1.4 REST APIs

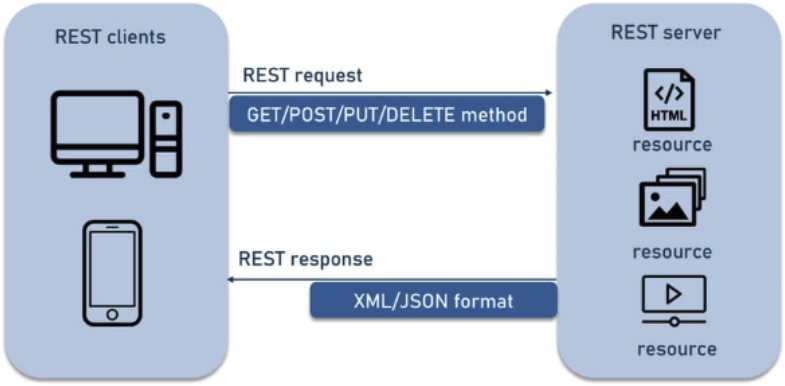


Figure 2.2: REST API communication overview, depicting a request and response exchange between a REST client and a server [6].

REST APIs communicate through HTTP requests to perform standard Create, Read, Update, Delete (CRUD) operations within a resource. When a request is made, the server authenticates the client and processes the request. The server then returns a response to the client, indicating if the request was successful or not. If it was successful, it transfers a representation of the state of the resource. An overview of this process can be seen in Figure 2.2. Although the information transferred between the call can have multiple formats, such as JSON, HyperText Markup Language (HTML), Python, or plain text, JSON is the preferred method because both humans and machines can read it, and it is programming language-agnostic [35].

For an API to be considered RESTful, it needs to follow the principles of the REST architecture, which are identified and explained below:

Uniform interface All API requests for the same resource should look the same, no matter where the request comes from [37].

Client-server decoupling Client and server applications must be completely independent of each other [37].

Statelessness Server applications are not allowed to store any data related to a client request [37].

Cacheability Resources should be cacheable on the client or server side [37].

Layered system architecture Neither the client nor the server can tell whether it communicates with the end application or an intermediary [37].

Code on-demand (optional) Apart from static resources, responses can also contain executable code that should only run on-demand [37].

When interacting with a RESTful API, the client must send a request composed of the following components:

URI The URI is what the server uses to identify each resource. For REST services, the server identifies resources using an Uniform Resource Locator (URL) [15].

Method The HTTP method that specifies what the server should do with the resource [15].

HTTP headers Used to exchange metadata between the client and the server, such as the format of the request and response and information about the request status [15].

Before the server can send a response, it needs to authenticate the request. This can be done using one of the following methods:

HTTP basic authentication The client sends the username and the password, both encoded with base64, in the request header [15].

HTTP bearer authentication The client sends the bearer token to access resources. A bearer token is an encrypted string of characters the server generates in response to a login request [15].

API keys The server assigns a unique value to a first-time client and, when the client tries to access a resource, it uses the key to verify himself [15].

OAuth The server requests a password and then asks for an additional token to complete the authorization process [15].

Once the request has been authenticated and processed, the server returns a response composed of the following components:

Status line The status line contains a three digit code that communicates request success or failure [15].

Message body Contains the resource representation in the format mentioned in the request header [15].

Headers Contains metadata about the response, such as information about the encoding and content type [15].

2.1.2 Synchronous APIs

A synchronous API is a pattern that refers to a type of communication between software applications where requests and responses are exchanged in real-time and synchronously [99]. When a client makes a request to a server, it waits for the response of that server before moving on [118]. To maintain this continuous connectivity, synchronous APIs rely on HTTP, gRPC or Transmission Control Protocol (TCP) [34]. The workflow of a synchronous API call can be seen in Figure 2.3 and can be split into three stages:

Request A client makes a request to a server to perform an action [118].

Processing The server processes the client request and prepares a response to send back. During this entire process, the client waits for this response [118].

Response The server sends a response back to the client. Once the client receives it, it is able to perform new requests [118].

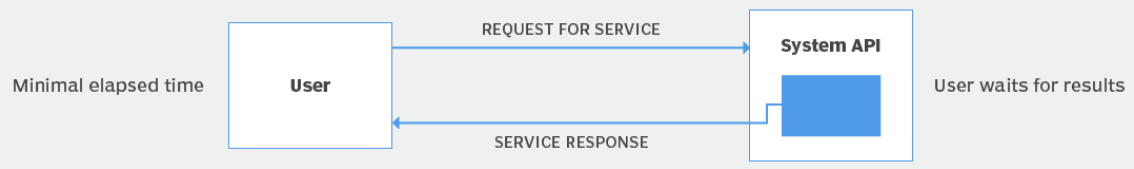


Figure 2.3: Synchronous API call, illustrating the communication between a user and a system API, where the user waits for the response before proceeding [61].

The OpenAPI specification, also known as OAS, and previously known as Swagger Specification, defines a standard, language-agnostic interface to HTTP APIs which allows both humans and computers to discover and understand the features of the service without the need to access the source code [126]. An OpenAPI Document is a single JSON or Yet Another Markup Language (YAML) document that conforms to the OAS. An OpenAPI Description (OAD) describes the surface of an API and its semantics. It can be made up multiple OpenAPI Documents. An OAD conforms to the OAS and must contain at least one *paths* field, *components* field, or *webhooks* field. An OAS can have the following objects:

Info Object API Metadata can be defined with the *info* object. This object has the following fields: (1) *title*, the title of the OpenAPI Document; (2) *version*, the version of the OpenAPI Document; (3) *summary*, a short summary of the API; (4) *termsOfService*, an URI for the terms of service for the API; (5) *contact*, the contact information of the API; (6) *license*, the license information of the API [126].

Paths Object The path holds the relative paths to the individual endpoints and their operations. A Path Item Object describes all operations available on a single path. Each path may include one or more endpoints mapped to HTTP methods, as well as shared parameters. The following fields can be specified in the Path Item Object: (1) *summary*, a summary that applies to all operations in the path; (2) *parameters*, a list of parameters that are applicable for all the operations described under the path [126].

Operation Object The operation object describes an individual endpoint. It describes the request and response formats, the parameters, security mechanisms, among other things. The following fields are some that can be specified in the operation object: (1) *summary*, a summary of what the endpoint does; (2) *parameters*, a list of parameters that are applicable for this endpoint; (3) *requestBody*, the request body of the endpoint; (4) *responses*, a list of possible responses the server may return [126].

Request Body Object The request body of an endpoint specifies the data that must be provided by the client to successfully execute the endpoint. It defines the structure and data types of the input payload. Examples and constraints among the parameters can also be specified using natural language. The request body can be defined with the following fields: (1) *content*, the content of the request body. This field is required; (2) *description*, a description of the request body. This field is not machine-readable; (3) *required*, determines if the request body is required in the request [126].

Response Object The response object of an endpoint specifies the data the server will return after processing the client's request. A HTTP status code is defined, as well as the structure and data types of the information returned. The response body can be defined with the following fields: (1) *description*, a description of the response. This field is required; (2) *content*, a map containing descriptions of potential response payloads; (3) *headers*, a map that associates the header name to its definition; (4) *links*, operation links that can be followed from the response [126].

Security Schema Object OAS currently supports security schemas that can be used by endpoints. Currently, OAS supports schemas for HTTP authentication, API keys, OAuth2 and mutual Transport Layer Security (TLS). Each security schema must specify the type being used in the field *type* [126].

2.2 State of the Art

This section presents a literature review on API contract validation using the PRISMA methodology, followed by a presentation and analysis of the state-of-the-art approaches and related tools. The literature review identifies the most relevant research in the field, while the state-of-the-art analysis covers the existing approaches for generating test cases and input data, specifying and validating constraints, and API linters.

2.2.1 Research Domain

With the evolution of software applications, the number of APIs currently being used is very large. Therefore, developers resort to API testing to make sure every API is functioning according to its specification. However, existing API contract validation techniques are either focused on generating test cases and input data to verify whether an API implementation conforms to its specification, or on linting the specification against a predefined set of rules to detect syntactic and structural issues.

The objective is to find existing automated API contract validation techniques that incorporate contextual or semantic understanding, extending API validation beyond traditional syntax correctness techniques. Figure 2.4 shows all the steps performed during this systematic review.

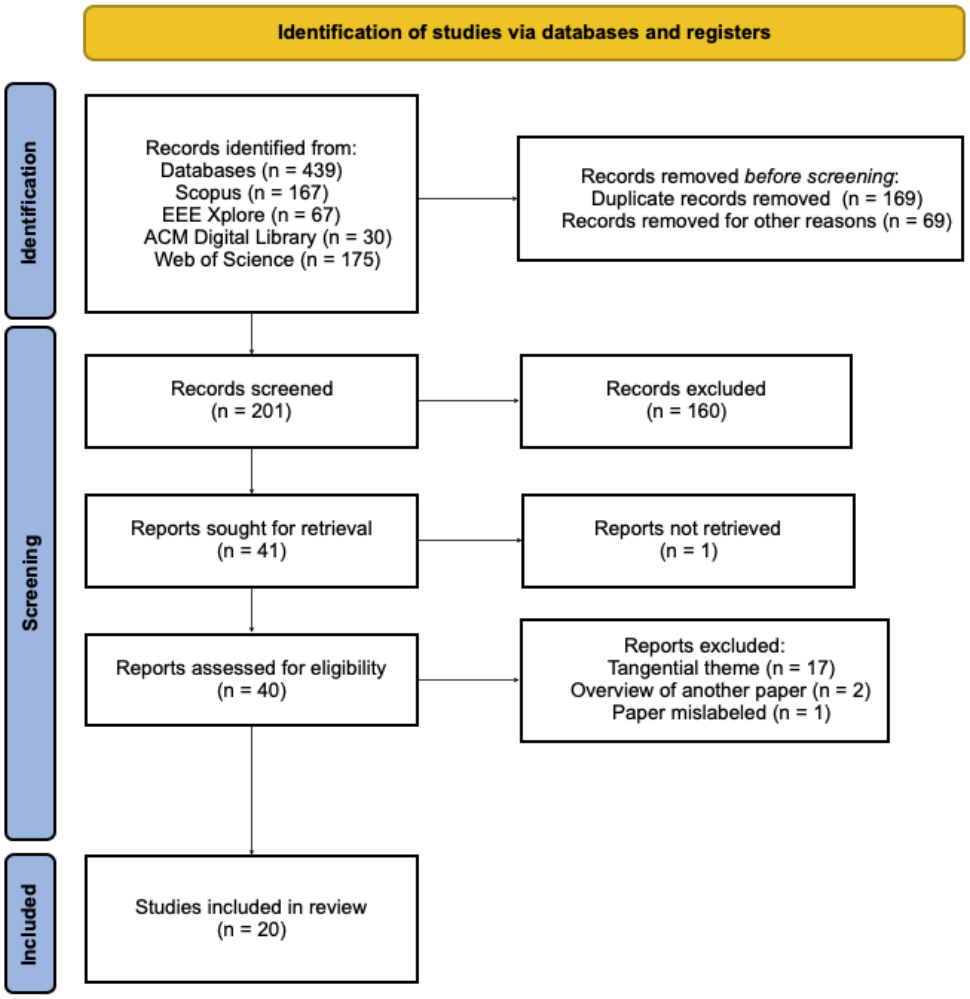


Figure 2.4: PRISMA flowchart depicting the systematic literature review process, from the identification of 439 records across four databases to the final selection of the 20 studies included in the review.

2.2.2 Identification Phase

To perform the literature review, an intensive and extensive search was performed on multiple online scientific databases, namely (1) Scopus[3], (2) IEEE Xplore[4], (3) ACM Digital Library[5] and (4) Web of Science[6].

To query these databases, a search string will be used to retrieve relevant pieces of literature that will be reviewed. A search string is a combination of keywords and boolean operators that can be applied to specific areas of documents, such as the title, abstract, authors, or keywords. The initial search string used can be seen in Table 2.1.

This search string can be decomposed into two parts:

[3] https://www.scopus.com/home.uri
[4] https://ieeexplore.ieee.org/Xplore/home.jsp
[5] https://dl.acm.org/
[6] https://www.webofscience.com/wos/woscc/smart-search

Table 2.1: Initial search string used on Scopus to retrieve relevant literature on API contract validation.

Identifier	Search String
QS1	*TITLE (API OR "Application Programming Interface") AND TITLE-ABS-KEY (test* OR validat* OR lint*)*

TITLE (API OR "Application Programming Interface"). This part retrieves every result that has at least one of the keywords in the document title.

TITLE-ABS-KEY (test* OR validat* OR lint*). This part searches the document title, abstract, and author keywords for one of the keywords *test**, *validat** and *lint**. These keywords use the * wildcard, so multiple variations of the same keyword can be searched for, such as, lint and linting.

In the end, these parts are combined together with the *AND* operator.

2.2.3 Refinement Process

The Scopus database was used as a reference in the refinement process of the search string. Refining the search string improves its efficiency by retrieving more relevant records and, at the same time, excluding irrelevant records.

2.2.3.1 QS1 Refinement Process

The initial search string *TITLE (API OR "Application Programming Interface") AND TITLE-ABS-KEY (test* OR validat* OR lint*)* proved to be too generic, retrieving 2927 results. To reduce the number of results, the search string was changed to *TITLE (API OR APIs OR "Application Programming Interface") AND TITLE-ABS-KEY ((API OR "Application Programming Interface") PRE/3 (test* OR validat* OR lint*) AND (WEB OR REST*))*. This change introduced the keyword *APIs* to the *TITLE* and the keywords *WEB* and *REST** to the textual content *TITLE-ABS-KEY*. Additionally, the operator *PRE/3* was also introduced in the textual content *TITLE-ABS-KEY*. The search string will now return results that have the keywords *API* or *"Application Programming Interface"* appearing within three words before the keywords *test**, *validat** and *lint**. With this refinement, the number of results was reduced to 160. In the third and final refinement, the keywords *contract*, *schema* and *specification* were added to the left side of the *PRE* operator, as these keywords are often used to refer to APIs. In the final version, the search string was able to retrieve an additional 7 results, increasing the total number of documents to 167. Table 2.2 shows the final search string used on Scopus. The search string used on every database can be seen in Table A.1.

Table 2.2: Final refined search string used on Scopus to retrieve relevant literature on API contract validation.

Identifier	Search String
QS1	*TITLE (API OR APIs OR "Application Programming Interface") AND TITLE-ABS-KEY ((API OR APIs OR "Application Programming Interface" OR contract OR schema OR specification) PRE/3 (test* OR validat* OR lint*) AND (WEB OR REST*))*

2.2.3.2 Results

In total, after querying each database with each refined search string, we were able to retrieve 439 results. Table 2.3 shows the distribution of the results obtained.

Table 2.3: Number of results retrieved using the refined search string across each database, resulting in 439 records.

	SCOPUS	IEEE Xplore	ACM Digital Library	Web of Science	Total
QS1	167	67	30	175	439

2.2.3.3 Search Criteria

The search criteria used in each database to retrieve the documents for screening were the following:

- Scopus: Only results published from 2017 onward and classified under "Computer Science" or "Engineering" subject areas were considered.
- IEEE Xplore: Only results published from 2017 onward were considered.
- ACM Digital Library: Only results published from 2017 onward were considered.
- Web of Science: Only results published from 2017 onward and classified under "Computer Science" or "Engineering" research areas were considered.

After applying these criteria, the number of total documents decreased by 69, from 439 to 370.

2.2.4 Screening Phase

After querying the online databases and applying the search criteria, all the retrieved records went through a screening phase. This is a manual process where each record is analyzed to evaluate whether the record is relevant or not.

To manage all the records, Zotero[7] was used. Zotero is a management tool that helps users collect and organize documents and their information.

[7] https://www.zotero.org/

2.2.4.1 Inclusion and Exclusion Criteria

In the screening phase, each record was analyzed and classified according to a predefined set of rules. These rules are the backbone of the screening phase, deciding whether or not a document is relevant. These rules can be divided and grouped into inclusion and exclusion criteria. Table 2.4 shows the inclusion and exclusion criteria used.

Table 2.4: Inclusion and exclusion criteria used to determine the relevance of each record during the screening phase.

Identifier	Criteria
C1	Publication originated from the databases.
C2	Publication date is from 2017 onward.
C3	Publication is written in English.
C4	Publication is an article, conference paper or book.
C5	Publication's content is relevant and focused in API contract validation.
C6	Publication's full-text is accessible.
C7	Publication is in its final version.
C8	Publication is not a duplicate.
C9	Publication is not retracted.
C10	Publication is not a tutorial.

2.2.4.2 Duplicate Removal

To detect and remove duplicates, Zotero's duplicate detection feature was used. Currently, Zotero compares the title, the Digital Object Identifier (DOI), the International Standard Book Number (ISBN), the year of publication, and the authors [18]. After using this feature, Zotero detected 164 duplicated publications, resulting in 206 publications. After a manual review, 5 more publications were identified as duplicated, bringing the number down to 201 unique publications.

2.2.4.3 Screened Records

In each of the 201 publications, the title and abstract were analyzed and classified according to the criteria presented in Table 2.4. Of the 201 unique publications, 41 were considered for a more in-depth analysis and 160 were considered not relevant.

2.2.5 Eligibility Phase

After a superficial analysis of each paper, a more in-depth analysis was conducted to evaluate if the paper is relevant in the context of API contract validation.

2.2.5.1 Full-text Analysis

Now that all screened records have been superficially analyzed, the full content of each publication was analyzed to determine if it is relevant to API contract validation. Of the 41 relevant publications, 21 were excluded for the following reasons:

- 17 papers were excluded because they focused on tangential themes to API contract validation.
- 2 papers were excluded because they were an overview of a paper already seen.
- 1 paper was excluded because the full-text was not accessible.
- 1 paper was removed because it was mislabeled as published after 2017, when it was published earlier than 2017.

2.2.5.2 Selected Records

In total, only 20 papers were considered relevant. A brief description of each paper can be seen in Tables 2.5, 2.6, 2.7, 2.8, 2.9, 2.10, where the research domain and the approach/domain are presented.

Table 2.5: Brief description of the five selected papers published in 2024, including their research focus and domain.

Reference	Research Purpose	Approach/Domain
Huynh et al. [36]	Address the limitations of current API testing tools, introducing an approach for mining and validating constraints within API specifications using Large Language Model (LLM).	API Constraints Mining, AI
Le et al. [82]	Present KAT, an AI-driven approach for dependency-aware automated API testing.	Endpoint Sequence-Based Testing, AI
Kim et al. [60]	Enhance API specifications by using LLMs to extract machine-interpretable rules from human-readable descriptions.	API Constraints Mining, AI
Yuncharoen and Suwannasart [132]	Generate RESTful API test scripts that account for database constraints using an OAS.	API Constraints Validation
Corradini et al. [20]	Leverage Deep Reinforcement Learning (DRL) to generate valid sequences of endpoints to test and valid input test data.	Generation of Test Scripts, AI

Table 2.6: Brief description of the five selected papers published in 2023, including their research focus and domain.

Reference	Research Purpose	Approach/Domain
Alonso, Segura, and Ruiz-Cortés [7]	Overcome REST APIs testing tools limitations by automating the generation of test oracles for REST APIs through the dynamic detection of likely invariants.	API Constraints Validation
Kim et al. [59]	Enhance API testing by leveraging information found in human-readable parts of OAS using Natural Language Processing (NLP).	API Constraints Mining, AI
Nguyen et al. [98]	Automate the creation of test scripts and data for API testing based on the API specification using LLMs.	Generation of Test Scripts, AI
Felício, Simão, and Datia [26]	Present a framework that uses a Domain-Specific Language (DSL) called Test Specification Language that allows the creation of tests focusing on the response correctness.	Generation of Test Scripts, DSL
Alonso et al. [8]	Generate realistic input data from OAS using NLP and semantic knowledge discovery.	Generation of Realistic Input Data

Table 2.7: Brief description of the three selected papers published in 2022, including their research focus and domain.

Reference	Research Purpose	Approach/Domain
Wu et al. [131]	Present RestCT, a systematic and automated tool that uses Combinatorial Testing (CT) and inter-parameter constraints to test RESTful APIs.	Endpoint Sequence-Based Testing, CT
Liu [87]	Extract parameter constraints from OAS descriptions by leveraging NLP techniques, such as Entity Recognition and Relationship Extraction.	API Constraints Mining, AI
Liu et al. [88]	Present Morest, a tool that creates tests based on a RESTful-service Property Graph (RPG) to model relationships between endpoints and schemas.	Endpoint Sequence-Based Testing

Table 2.8: Brief description of the two selected papers published in 2021, including their research focus and domain.

Reference	Research Purpose	Approach/Domain
Martin-Lopez et al. [96]	Present a DSL for the specification of inter-parameter dependencies to address the limitations of API specification languages.	Specification of API Constraints, DSL
Mirabella et al. [97]	Validate API requests using a deep learning-based approach.	Validation of API Constraints, AI

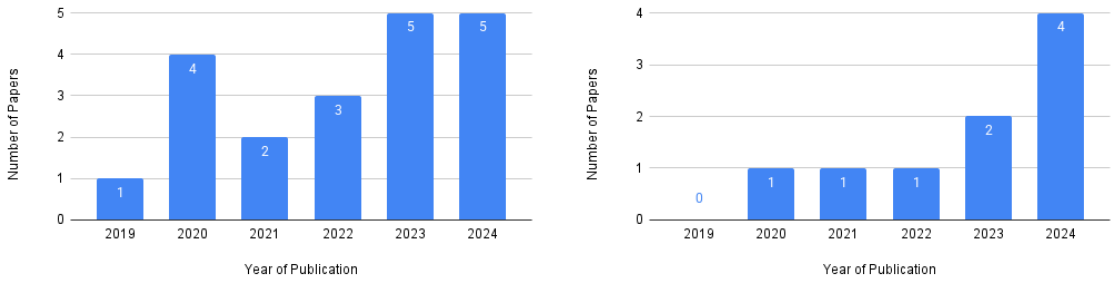
Table 2.9: Brief description of the four selected papers published in 2020, including their research focus and domain.

Reference	Research Purpose	Approach/Domain
Martin-Lopez, Segura, and Ruiz-Cortés [94]	Address the limitation of testing inter-parameter dependencies by presenting a framework for automated black-box testing.	Validation of API Constraints
Viglianisi, Dallago, and Ceccato [129]	Automatically generate test cases for APIs based on the API specification and an Operation Dependency Graph (ODG) to represent dependencies among the API's operations.	Endpoint Sequence-Based Testing
Bahrami et al. [16]	Combine several machine-learning algorithms to extract key attributes from API documentation and generate a machine readable OAS.	API Constraints Mining, AI
Karlsson, Čaušević, and Sundmark [57]	Leverage Property-based Testing (PBT) to generate test cases and input data.	Generation of Test Scripts, PBT

Table 2.10: Brief description of the one selected paper published in 2019, including its research focus and domain.

Reference	Research Purpose	Approach/Domain
Atlidakis, Godefroid, and Polishchuk [14]	Present a stateful fuzzing tool which generates sequences of test cases based on dependencies inferred from an OAS.	Endpoint Sequence-Based Testing

2.2.6 Results and Discussion



(a) Number of selected papers published by year, from 2019 to 2024.

(b) Number of selected papers that use AI per year, from 2019 to 2024.

Figure 2.5: Distribution of the 20 selected papers by year of publication, showing the total number of papers per year (left) and the number of papers that use AI per year (right).

Now that a brief description of the 20 papers have been presented, some statistical results of the papers will be presented and discussed. These papers can essentially be divided into 2 groups, one that focus on mining and validating API constraints from API specifications, while the other focuses on the generation of test cases based on the API specification.

Of the 20 papers, half of them [7, 16, 36, 59, 60, 87, 94, 96, 97, 132] can be inserted into the first group. [16, 36, 59, 60, 87] talk about mining and inferring constraints from API specifications and documents, [7, 94, 97, 132] about validating whether API follow the imposed constraints and [96] present a DSL to specify constraints in the API specification.

The other half [8, 14, 20, 26, 57, 82, 88, 98, 129, 131] can be inserted in the second group. [14, 82, 88, 129, 131] talk about endpoint sequence-based testing, where a sequence of endpoints is defined to test the System Under Test (SUT), [20, 98] use AI as a core component, [26] present a DSL to configure API tests, and [8, 57] use other approaches, specifically an approach for generating realistic input data and an approach that uses CT.

In Figure 2.5a, we can see that there has been an increase in research in the context of API contract validation over the last years. Despite this increase, this area of research is still not widely explored, as the average number of papers published per year is around 3 per year.

In Figure 2.5b, we can see an emerging trend over the last years, which is the usage of AI in API contract validation. Of the 20 analyzed papers, 9 papers [16, 20, 36, 59, 60, 82, 87, 97, 98] use AI as a core component. This trend can be explained by the high ability current AI tools have to process natural language, which makes AI suitable for the analysis of API specifications, allowing the tool to infer hidden dependencies among API endpoints and API parameters. This ability can also be used alongside code generation to automatically generate test cases and test data from the descriptions of API endpoints. These descriptions are written in natural language and are not machine-readable, and often contain important details about the usage of the endpoint, such as parameter restrictions and parameter values examples. As a result, AI tools can be used to automatically generate test cases and test input data that comply with the specified restrictions.

2.2.7 Constraint-Based API Testing

Current API specifications describe how an API should be used, e.g., the endpoints, their parameters, and the expected result. However, they cannot formally describe dependencies/constraints between the data. A study from Martin-Lopez, Segura, and Ruiz-Cortés [93] showed that 85% of APIs have constraints among their input. These constraints can be divided into the following categories:

Dependencies among Operations. This type of dependency exists when the execution of an endpoint depends on the execution of another endpoint. For example, if we want to book a flight, we need its *id*. However, we can only obtain this *id* if the result of another endpoint (e.g. *GET /flights*) successfully returns all the available flights for booking [82].

Dependencies among Parameters. This dependency exists when parameters of the same endpoint have constraints. For example, if we have an endpoint that allows us to book a flight, and it receives two parameters, the departure date and the arrival date, it is expected that the departure date precedes the arrival date [82].

Dependencies between An Operation and Its Parameters. This dependency exists when an endpoint imposes constraints on its parameters. For example, the departure date and the arrival data of a flight must be in the future [82].

The lack of ability to represent these constraints, such as value ranges, conditional parameters, and dependencies among input data and/or endpoints, forces developers to describe such dependencies as part of the description of the parameters in natural language, which is not machine-readable and may result in ambiguous or incomplete descriptions [96]. Moreover, API validation is deeply affected because the probability of generating test cases and input data that satisfy all these constraints is very low.

2.2.7.1 DSL for the Specification of API Constraints

Martin-Lopez et al. [96] present a DSL named Inter-parameter Dependency Language (IDL). This is a textual DSL focused on expressing inter-parameter dependencies and not the parameters themselves, which allows the DSL to be easily integrated with existing API specification tools, like the OAS. The following types of inter-parameter dependencies are supported:

Requires. The presence of a parameter p_1 requires the presence of another parameter p_2. This type of dependency is expressed as *"IF* p_1 *THEN* p_2*"*.

Or. Given a set of parameters $p_1, p_2, \ldots, p_n$, at least one of these parameters must be included. This type of dependency is expressed as *"Or(*p_1, p_2, p_3*)"*.

OnlyOne. Given a set of parameters $p_1, p_2, \ldots, p_n$, only one of these parameters must be included. This type of dependency is expressed as *"OnlyOne(*p_1, p_2, p_3*)"*.

AllOrNone. Given a set of parameters $p_1, p_2, \ldots, p_n$, either all of the parameters are included or none of them are included. This type of dependency is expressed as *"AllOrNone(p_1, p_2, p_3)"*.

ZeroOrOne. Given a set of parameters $p_1, p_2, \ldots, p_n$, at most one parameter may be included. This type of dependency is expressed as *"ZeroOrOne(p_1, p_2, p_3)"*.

Arithmetic/Relational. Given a set of parameters $p_1, p_2, \ldots, p_n$, these parameters must satisfy an arithmetic or relational constraint. This type of dependency is expressed using relations such as $"p_1 < p_2"$ or arithmetic expressions such as $"p_1 + p_2 == 100"$.

Complex. This dependency involves two or more of the constraints previously presented. An example of this dependency could be *"Or(p_1, p_3 AND $p_5, p6 ==$ 'B')"*.

An OAS can be extended to support IDL dependencies through an OAS extension called IDL4OAS. In each API endpoint, a new field called `x-dependencies` can be added where the IDL dependencies are defined. Figure 2.6 shows an example of an extended OAS. Martin-Lopez et al. also propose IDLReasoner, an analysis tool for extended OAS files. This tool transforms API endpoints' parameters, their domains, and the IDL dependencies to a MiniZinc file, a high level constraint modeling language. The IDL dependencies are then parsed and translated into MiniZinc constraints. The tool then returns a boolean indicating whether there exists at least one request that satisfies all dependencies.

```yaml
paths:
  /example/route:
    get:
      operationId: exampleOperation
      parameters:
        - in: query
          type: boolean
          required: false
          name: p1
        - in: query
          type: integer
          required: false
          name: p2
        - in: query
          type: string
          required: false
          name: p3
      x-dependencies:
        - Or(p1, p2, p3);
        - IF p1==true THEN p2;
```

Figure 2.6: OAS extended with IDL dependencies, showing an example endpoint with three query parameters and two inter-parameter dependency rules defined using the `x-dependencies` extension [95].

2.2.7.2 Mining and Inferring API Constraints

Huynh et al. [36] introduced APITesting, an approach to mine constraints from response bodies and generate test cases to validate these constraints. This approach uses search heuristics to extract descriptions related to endpoints, parameters, and response properties. These descriptions then serve as input for an LLM that identifies constraints from the descriptions. The identified constraints are then fed into an LLM, which will generate test cases to validate these constraints.

Liu [87] introduces RESTInfer, an approach to infer parameters constraints by natural language processing. RESTInfer starts by identifying parameters in the description of the API specification using a Recurrent Neural Network (RNN). This RNN uses NERSE, a framework for Named-Entity Recognition (NER) that was specifically trained to recognize parameter entities within the context of software engineering [128]. RESTInfer then uses a parser created by Chen and Manning [19] to extract relationships, which are then used to generate parameter constraints in the Backus-Naur Form (BNF). An overview of the BNF grammar and RESTInfer is shown in Figures 2.7 and 2.8, respectively.

```
⟨constraint⟩ ::= ⟨param⟩ ⟨op⟩ ⟨param⟩
                 | ⟨constraint⟩ and ⟨constraint⟩
                 | ⟨constraint⟩ or ⟨constraint⟩
        ⟨op⟩ ::= = | < | > | and | xor
     ⟨param⟩ ::= param_name | constant
```

Figure 2.7: BNF grammar used by RESTInfer for representing parameter constraints, defining the structure of the constraints, operators, and parameters [87].

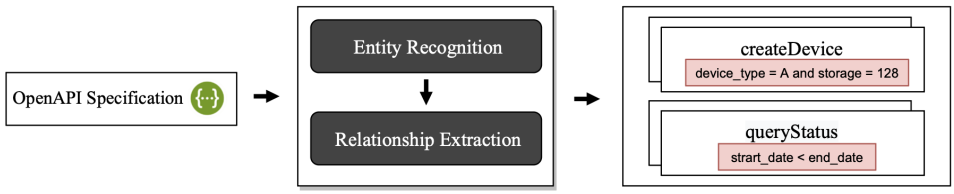


Figure 2.8: Overview of the RESTInfer approach, depicting the extraction of parameter constraints from an OAS through entity recognition and relationship extraction, producing endpoint-specific constraints in BNF [87].

Kim et al. [59] propose NLPtoREST, an approach that enriches OAS from human-readable parts of OAS. Figure 2.9 shows an overview of NLPtoREST. NLPtoREST starts by parsing the OAS to extract its human-readable and machine-readable parts. The human-readable part is then analyzed to extract rules. A custom pre-trained NLP model, which was trained on parameter descriptions taken from OpenAPI documents, is used to identify terms that, if present, may indicate potential testing rules, such as parameter constraints, parameter examples, and inter-parameter dependencies. If a term is identified, a constituency parse tree is built. An algorithm then searches for a value in this constituency parse tree. A constituency parse tree represents the grammatical

structure of a sentence in a tree format. Part-of-Speech (PoS) tagging is used to assign a grammatical role to each word in the sentence, and sentence constituents are used to describe the structural components of a sentence, such as phrases or clauses. An example of a constituency parse tree can be seen in figure 2.10.

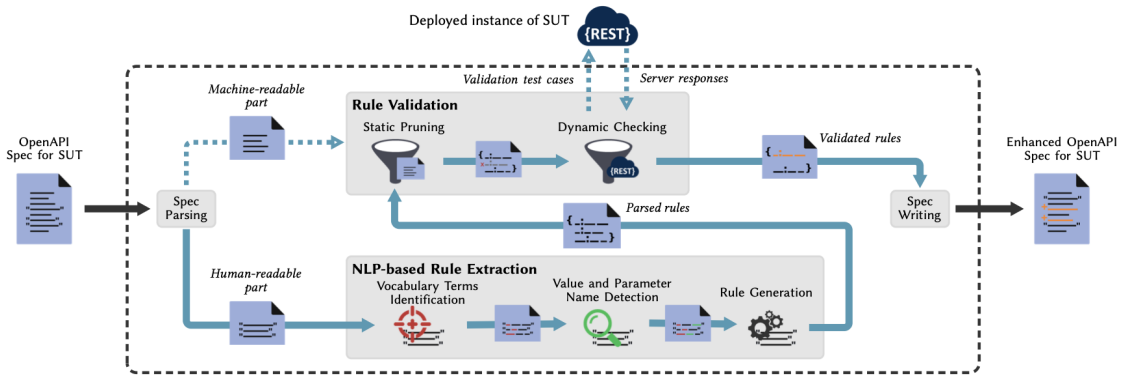


Figure 2.9: Overview of the NLPtoREST approach, depicting the pipeline from an input OAS through an NLP-based rule extraction and rule validation to produce an enhanced OAS with additional parameter constraints [59].

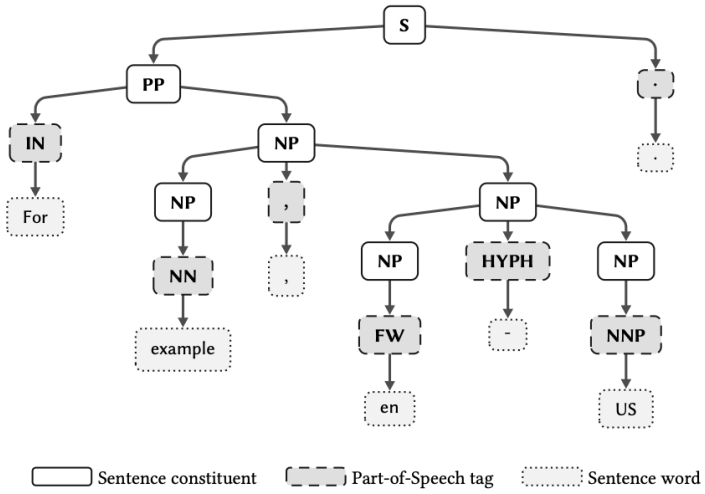


Figure 2.10: Constituency parse tree of the sentence “For example, en-US”, illustrating how sentence constituents, PoS tags, and sentence words are structured hierarchically [59].

Lastly, OpenAPI rules are generated based on the information derived in the previous stages. Identified vocabulary terms and their corresponding values are combined to produce rules using IDL (Section 2.2.7.1). A rule validation step is then performed, consisting of a static-pruning phase that discards syntactically incompatible rule combinations and a dynamic checking phase that validates the generated rules against the SUT. The validated rules are then added to the original OAS.

Kim et al. [60] introduce RESTGPT, an approach that leverages LLMs to enhance API specifications by identifying constraints and generating example parameter values. RESTGPT starts by parsing the OAS, identifying both human-readable and machine-readable parts. The former is then given to a Rule Generator, which will prompt an LLM (GPT-3.5 Turbo) to generate rules. To achieve a better result, each prompt is designed around four core components: (1) Guidelines, which form the foundation of the model, defining its perspective and primary objectives; (2) Cases, which decompose rule extraction into smaller, manageable steps to minimize ambiguity and improve the model's processing abilities; (3) Grammar Highlights, which define the key operators and the vocabulary the model must recognize and apply during rule extraction; (4) Output Configuration, which defines the output format to organize the extracted information into a simple structure suitable for processing. The generated rules are then incorporated into the original OAS, creating a new enhanced specification.

Bahrami et al. [16] introduces an approach to generate a machine-readable OAS based on key attributes from API documentation. A model will extract API features from HTML table tags and predict an OAS type for each extracted HTML table, such as parameters, responses and endpoints. To avoid false-positive classifications, another model is used to recognize different API features according to the document structure and not the textual content. A Convolutional Neural Network (CNN) is then used to map individual elements from HTML tables (e.g. table columns or values) to an OAS object.

2.2.7.3 Validation of API Constraints

Mirabella et al. [97] proposes the use of deep learning, more specifically a Multilayer Perceptron (MLP), to automatically infer whether an API request is valid or not without calling the actual API. In this approach, a data collection with labeled API calls is needed. An API call is labeled as *valid* if it meets all parameter constraints indicated in the API specification and returns a successful response (*2XX* status code), and is labeled as *invalid* if it returns a client error response (*4XX* status code). This way, we ensure that the invalid API calls did not fail because they did not respect its constraints. After obtaining the dataset, a data preprocessing step is performed to handle empty values and string inputs. The model accepts the parameters of an API call as input and returns the predicted validity as output. Once the system has been trained, the validity of APIs can be predicted by passing the arguments into the model and by verifying whether the result is *valid* or *invalid*.

Martin-Lopez, Segura, and Ruiz-Cortés [94] present RESTest, a framework for automated black-box testing of RESTful APIs. RESTest supports the specification and automated analysis of inter-parameter dependencies using the IDL tool suite. RESTest uses the extended OAS with IDL rules as input. Then, a test model is automatically generated based on the provided OAS. This test model contains all configuration settings related to testing the API and can be manually configured. The test model is then used with the OAS to generate abstract test cases, which are platform-independent. Each generated abstract test case is composed of its test inputs, expected output, and the information necessary to build the API request. If IDL rules are present, these will

be given to IDLReasoner, which will verify if the generated abstract tests respect the constraints specified by the IDL rules. The abstract test cases are then transformed into platform-specific executable test cases, which will be executed and validated based on the returned status code. Figure 2.11 shows an overview of RESTest.

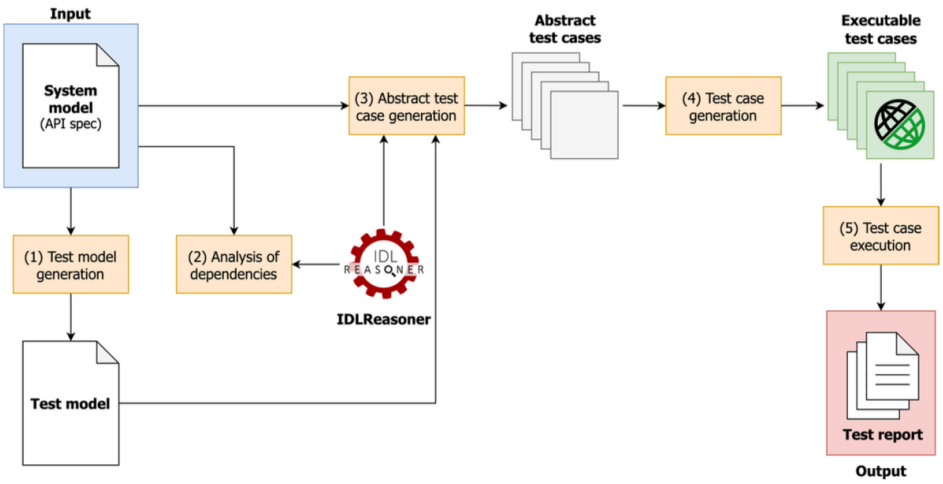


Figure 2.11: Overview of RESTest framework, depicting the five-step pipeline from an OAS through the test model generation, dependency analysis using IDLReasoner, abstract and executable test case generation, to test execution and report generation [94].

Alonso, Segura, and Ruiz-Cortés [7] present AGORA, an approach for automatic generation of test cases for REST APIs through the detection of invariants. An invariant is a property of the output that must always hold true. AGORA uses Beet, a software tool that takes as input an OAS, a collection of API requests, and their associated responses to generate two files, one that describes the structure of the API's input and output, and another one that captures the concrete values during the execution of each API call. These two files are then used by Daikon, a tool that detects possible invariants which can be used as test oracles.

Yuncharoen and Suwannasart [132] propose an approach for generating RESTful API tests scripts based on database constraints using an OAS. The OAS is parsed to extract information about the endpoints, request and parameter data, and response data. Then, information about the database tables, their columns and table constraints is retrieved from the database. This information is then mapped to the OAS data to determine which table constraints relate to specific OAS field. Test cases are then generated to validate or violate the database constraints.

2.2.8 Automated Test Case and Input Generation for REST APIs

Automated test case and input generation for REST APIs is a related field to API contract validation. Rather than validating the specification itself, these approaches focus on automatically generating test cases and input data based on the API specification, with the goal of verifying whether the API implementation conforms to its contract.

2.2.8.1 DSL for the Configuration of Test Cases

Felício, Simão, and Datia [26] present RapiTest. RapiTest combines automatic test generation from API specifications with a DSL named Test Specification Language (TSL), which is written in YAML in an easy-to-read file for both humans and machines. TSL allows for the specification of workflows. Each workflow is composed of a sequence of multiple tests, the validation process of the sequence, and the configuration of how the sequence will be executed. Both OAS and TSL files are then validated and serialized into a final JSON file that represents the complete test suite. This file is then deserialized to execute every test against the SUT, generating a new JSON file with the results.

2.2.8.2 Endpoint Sequence-Based API Testing

Le et al. [82] present KAT, an AI-driven approach to autonomously generate test cases to validate APIs. KAT starts by leveraging LLMs, specifically GPT, to identify dependencies between API endpoints. These dependencies are represented using an ODG. An ODG is a directed graph $G = (N, V)$, where each node N represents an endpoint within the API. The presence of an edge $v \in V$, articulated as $v = n_1 \rightarrow n_2$ represents a dependency between nodes n_1 and n_2. An example of an ODG can be seen in Figure 2.12. From this ODG, we can infer multiple sequences of execution of API's endpoints in which one request prepares the data (parameters and request body) needed for succeeding API requests. GPT is then used to generate valid and invalid test data as a result of its high ability to understand natural language and infer constraints without manual intervention. Lastly, GPT generates test scripts based on the sequences and test data previously generated.

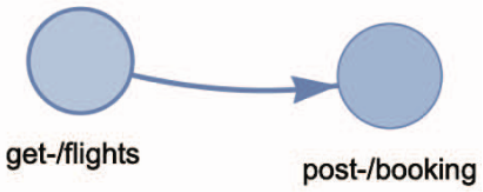


Figure 2.12: ODG representing a dependency between `GET /flights` and `POST /booking`, where the booking operation depends on the result of the flights operation [82].

Atlidakis, Godefroid, and Polishchuk [14] present RESTler, a stateful fuzzing tool which analyzes an OAS, infers dependencies among request types and generates test cases. RESTler starts by analyzing the OAS, where it infers dependencies among API endpoints and constructs a test-grammar with the required information to generate an API request and to process the response of the request. RESTler then starts sending sequences of API requests and analyzes the HTTP status codes returned by the responses. The sequence starts with a single API request and, as long as the response is valid, the sequence keeps growing by adding other endpoints to the sequence.

Viglianisi, Dallago, and Ceccato [129] present RestTestGen, an approach to automatically generate test cases for REST APIs. RestTestGen starts by computing an ODG to represent dependencies among the available endpoints from the OAS. Test cases are then generated using both

the OAS and the ODG, where the latter is used to sort the operations to test depending on their data dependencies. Input values are generated probabilistically, where the input can be randomly generated or retrieved from a response dictionary containing the values from every successfully tested API request. Each successful test is then mutated to violate data constraints to stress the data validation features of the API.

Wu et al. [131] propose RestCT, a systematic and automatic approach that adopts CT to test RESTful APIs. RestCT starts by parsing an OAS to extract every available endpoint and to automatically infer dependencies between endpoints based on their resource hierarchies and CRUD semantics, which constrain the generation of endpoint sequences. These dependencies are then used along with CT to generate a set of endpoint sequences to determine the execution order of the endpoints. For each endpoint sequence, CT is used to generate valid combinations of input parameters values, resulting in executable test cases.

Liu et al. [88] present Morest, a RESTful API testing technique based on a dynamically updated RPG. Morest uses this RPG to capture relationships between API endpoints and their interactions with data schemas. An algorithm is then used to visit every schema in the RPG and collect the endpoints related to the schema to construct API call sequences. These call sequences are then filtered to remove the endpoints that violate CRUD semantics. Input data is generated randomly for independent endpoints, while dependent ones reuse responses from previous API calls. The RPG is dynamically updated based on the execution feedback from the API calls, allowing to refine its understanding of the API over time.

2.2.8.3 AI-Based Approaches

Corradini et al. [20] introduce DeepREST, an approach for automatically testing REST APIs. DeepREST trains an intelligent agent to learn and optimize a strategy to test REST APIs and consists of two components: (1) Curiosity-Driven State Explorer and (2) Experience-Driven Input Generator. The first component is a DRL agent that selects the next API endpoints to execute based on the current state of the API. The state of the API can be described as a list of integers, where each integer corresponds to an API endpoint and its value is the number it has been successfully tested. Using a curiosity-driven strategy, the agent is positively rewarded when it discovers new API states. The second component is composed of multiple reinforcement learning agents which, based on their experience, generate test input data. These two components then work together to make API requests to the SUT. If an API request is validated, DeepREST proceeds to make new API requests with small mutations in the parameter values of the validated API request, promoting input diversity, resulting in increased test coverage.

Nguyen et al. [98] propose an approach to generate test scripts and test data for API testing using LLMs, specifically GPT. An OAS is analyzed to extract information about the API's endpoints. This information is then used to generate multiple prompts to the LLM, which will generate test cases and input data for each endpoint against multiple scenarios. To achieve better results, each prompt is structured around five core components: (1) Test case scenario; (2) Data generation rules; (3) Extracted information from the OAS analysis; (4) Expected result template;

(5) Execution feedback, when available. The Katalon testing tool is used to execute the generated API calls and for generating logs for each call. These logs will then be used as feedback to improve the generation of future test cases.

2.2.8.4 Other Approaches

This section presents additional approaches to automated test case and input generation for REST APIs that do not fit in the previous categories, covering realistic input data generation and PBT.

Alonso et al. [8] present ARTE, an approach for the automated generation of realistic test input data. ARTE starts by collecting the name of the parameters, description and constraints from the OAS. Then, a keyword is generated for the target parameter based on its name and description. A SPARQL query is then built for each keyword to obtain a list of predicates, which are then used to extract realistic input data.

Karlsson, Čaušević, and Sundmark [57] propose an approach to generate test from OpenAPI documents using PBT. The OpenAPI document is parsed and represented in an format more suitable for further processing. During this process, Clojure.spec definitions are attached to each parameter and response definition. These definitions are then used to construct generators, which are responsible for automatically produce test cases, and to validate response payloads and status codes. For tests that depend on the system state or on previous API calls, the tool stores previous responses from successful requests so that the data can be reused in subsequent requests.

2.2.9 Related Work

This section presents tools that were not identified in the literature review but were considered relevant to the problem addressed in this dissertation, as they represent widely adopted approaches for OpenAPI Specification validation.

2.2.9.1 Static-Based Linters

Static-based linters are tools that validate API specifications through the application of a predefined set of rules. These tools, such as Spectral[8], Zally[9], and IBM OpenAPI Validator[10], analyze the API specification and flag any violation that goes against the rules. These rules are manually defined in a configuration file, generally with a specific DSL, which allows developers to adapt them to their validation process. However, these rules are deterministic, limiting the range of issues that can be detected, from syntactic errors to structural issues, such as verifying missing required fields, naming convention violations, and other well-defined issues. Figure 2.13 shows an example of a Spectral rule that verifies whether query parameter names follow the camel case naming convention and do not contain digits.

[8] https://stoplight.io/open-source/spectral
[9] https://opensource.zalando.com/zally/
[10] https://github.com/IBM/openapi-validator

```
query-parameters-camelcase:
  given: '$..parameters[?(@.in == ''query'')].name'
  then:
    function: casing
    functionOptions:
      type: camel
      disallowDigits: true
  message: Query parameters should be camelCase and not contain digits
  severity: error
```

Figure 2.13: Example of a Spectral rule that verifies whether query parameter names follow the camel case naming convention and do not contain digits.

While static-based linters are effective for well-defined, pattern-based checks, they present significant limitations when it comes to detecting issues that require semantic reasoning. Since their validation process relies on explicitly defined rules, they do not have the capacity to detect issues that depend on reasoning over both OpenAPI Specification and external guidelines. Additionally, the fact that rules have to be manually defined in a configuration file can be a time-consuming activity to both write and maintain. As organizational guidelines evolve, the ruleset must be continuously updated, creating a form of technical debt that grows with the scale of the guidelines and the API collection.

2.2.9.2 Context-Aware Linters

LintGPT[11] is an API specification linter that uses AI to analyze API specifications. This tool, instead of using DSLs to define well-defined rulesets, defines constraints in natural language that will be sent to an LLM. The LLM will then analyze the API specification and give feedback based on the constraints [125].

Although this tool introduces a novel concept of being aware of the surrounding context, it has several limitations, such as the use of a public LLM, which many companies do not allow to use due to security and privacy reasons. Additionally, LintGPT requires the user to manually define each constraint, which can be a time-consuming and prone to error task when internal policies or guidelines evolve, and existing prompts need to be updated, either by creating new ones, or by changing or deleting the current ones. Furthermore, LintGPT processes the entire specification at once, as well as all of the constraints, which is not scalable due to the limited context size of LLMs. This limitation becomes particularly problematic when dealing with large API specifications or extensive organizational guidelines, as the volume of the content may exceed

[11] LintGPT was acquired by Atlassian in April 2024 and is no longer actively maintained [119].

the context window, increasing the risk of incomplete or inaccurate responses. Beyond this, since LintGPT only validates against constraints manually defined by the user, it cannot detect issues that were not explicitly defined.

2.2.10 Analysis of State of the Art

Current state of the art approaches can be divided into two different groups: constraint-based API testing and automated test generation for APIs. Both groups focus on improving API contract validation by addressing current limitations of API specifications, such as the inability to specify dependencies between operations and parameters.

Constraint-based API testing addresses the limitation that API specifications cannot specify API dependencies. Several approaches have been proposed, ranging from the creation of a DSL to define these dependencies, the usage of AI to process and analyze natural language from human-readable description fields to infer dependencies, to validation frameworks that check whether API requests comply with these dependencies.

Automated test generation focuses on producing test scripts and/or input data while respecting API's dependencies, such as creating a resource before deleting it. These dependencies can be modeled through ODGs and RPGs. AI can also be used to generate sequences of endpoints, test data and test scripts.

In addition to these two groups, the related work presented commonly used tools for OpenAPI Specification validation. Static-based linters, such as Spectral, rely on a set of predefined rules, but can only detect syntactic and structural issues, falling short in the detection of issues that require semantic reasoning. LintGPT, a context-aware linter, explores the usage of LLMs to analyze API specifications and provide feedback by leveraging contextual information expressed in natural language. However, it is not scalable as it processes both the specification and constraints at the same time, which becomes problematic when dealing with large specifications or extensive guidelines, as the volume of the context may exceed the LLM's context window. Furthermore, LintGPT is limited to detecting only the constraints explicitly defined by the user. Additionally, both types of API linters are not fully automated and require a user to manually define each constraint, which can be a time-consuming and error-prone task, particularly as organizational guidelines evolve.

Overall, current research is focused on handling implicit API constraints based on the API specification, or on generating tests that automatically test the SUT based on the API specification. While static linters provide efficient and reliable validation of syntactic and structural issues, they lack the ability to reason about the semantics of an API. LintGPT shows how AI can be used to assist the specification analysis. However, this approach is not fully automated and still requires human intervention to maintain the queries it uses. Thus, a research gap exists in the context of API contract validation, specifically in the development of a fully automated, scalable, and context-aware solution capable of validating API specifications against external organizational guidelines, while also detecting semantic and context-dependent issues that existing static and context-aware tools are incapable of identifying.

2.3 Summary

This chapter presented all of the essential concepts and knowledge required for this dissertation, describing what a Web API is, its lifecycle, and the multiple types of Web APIs and architectural styles, with a particular focus on REST APIs. Additionally, synchronous APIs were presented with the OpenAPI Specification described in more detail.

A systematic literature review on API contract validation was then conducted using the PRISMA methodology. All the steps performed during the literature review were presented, from the research query and its refinement, and the selected databases where the articles were retrieved. The criteria used to define whether or not a paper was selected were also presented. A brief description of each selected article was also presented. It was possible to see that research on API contract validation is still a growing topic and has been increasing over the last years. The usage of AI has also been increasing in the context of API contract validation due to its high capability to process natural language and generate code. In addition, it was possible to see that current research focuses on exploring tools for mining API constraints and validating API constraints based on API specifications, or by generating test cases and input data based on the API specification.

Additionally, this chapter presented an overview of each state-of-the-art approach, as well as an overview of the relevant related work. An analysis of the overall state-of-the-art was also performed to compare existing solutions to identify their common characteristics and limitations. This analysis showed that, although the usage of AI has been increasing in this domain, existing approaches remain limited in both automation and context awareness. Through this analysis, it was possible to identify a research gap: the need for an automated and context-aware tool to validate APIs specifications.

Chapter 3

NysusAI

This chapter presents NysusAI, a scalable context-aware tool capable of validating an OpenAPI Specification against external guidelines, as well as detecting inconsistencies within the specification itself.

Section 3.1 defines the desired properties and capabilities of NysusAI. Section 3.2 presents the context used in this dissertation, as well as the OpenAPI Specifications dataset. Section 3.3 presents the architecture of NysusAI and its details. A summary of what was done in this chapter is presented in Section 3.4.

The codebase for NysusAI is publicly available on GitHub [123].

3.1 *Desiderata*

Based on the identified research gap and limitations of existing approaches, the following desired properties and capabilities were defined:

- **Automated Guideline Retrieval**: NysusAI must automatically retrieve, parse, process, and store the external guidelines without requiring manual intervention;
- **Automated Guideline Updates**: NysusAI must automatically update its stored guidelines so it is always up-to-date compared to its external source;
- **Context-Aware API Specification Validation**: NysusAI must interpret the stored guidelines and use that context to identify potential violations in the OpenAPI Specification.
- **Inconsistency Detection**: NysusAI must detect internal inconsistencies within the OpenAPI Specification;
- **Report Generation**: NysusAI must present all the issues found to the user in a structured format;
- **Scalability**: NysusAI must handle large OpenAPI Specifications, as well as a large set of guidelines;

3.2 Tool Context

Although this dissertation is developed in partnership with Kuehne+Nagel, the company's guidelines are not accessible to the public. For this reason, in this dissertation, the Zalando RESTful API and Event Guidelines[1] will be used instead, as they provide a comprehensive and publicly set of RESTful API design guidelines. Regarding the OpenAPI Specifications dataset, a total of 50 specifications were collected and used for both the development and evaluation of NysusAI.

3.2.1 Zalando RESTful API and Event Guidelines

Created by Zalando Open Source[2], these guidelines aim to ensure that an API specification is consistent, promoting interoperability and making it easier for clients to understand and use them more efficiently [120, 121]. These guidelines are composed of two categories: RESTful API guidelines and Event guidelines. Since this dissertation focuses on REST APIs, only the RESTful API guidelines will be used. Composed of 115 guidelines, these provide recommendations on how to design an OpenAPI Specification, covering topics such as *Meta information*, *Data Formats*, and *HTTP requests*, as well as operational and organizational recommendations, such as deprecation policies, client communication, and API usage monitoring. Each guideline is composed of a title, an ID, and a description explaining the purpose and the requirements of the guideline. An example of a guideline can be seen in Figure 3.1. Throughout this dissertation, the terms *rule* and *guideline* are used interchangeably, as are *Zalando guidelines* and *Zalando RESTful API guidelines*, both referring to the same concept.

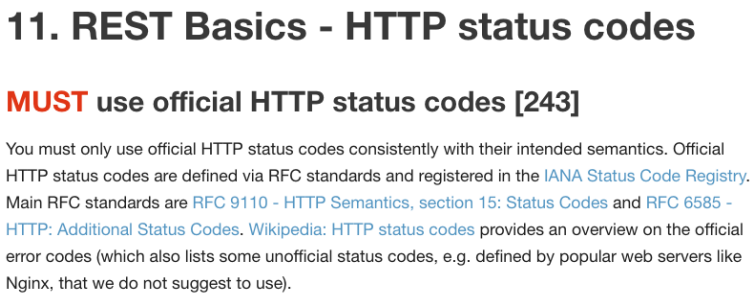


Figure 3.1: Example of a Zalando guideline, showing guideline #243 that requires the use of official HTTP status codes, under the *HTTP status codes* section [120].

3.2.2 OpenAPI Specifications Dataset

In order to obtain a realistic dataset, all of the 50 OpenAPI Specifications are from real-world companies whose specifications are available to the public: (1) Adyen[3], (2) Amadeus[4], (3) Brex[5], (4)

[1] https://opensource.zalando.com/restful-api-guidelines/
[2] https://opensource.zalando.com/
[3] https://www.adyen.com/
[4] https://amadeus.com/en
[5] https://www.brex.com/

Countrylayer[6], (5) DHL[7], (6) Youtube[8], (7) Kong[9], and (8) Kuehne+Nagel[10]. Table 3.1 shows the number of specifications used by company, as well as their domains. Adyen and Youtube specifications were obtained from API.guru[11], an open repository that aggregates publicly available API specifications from companies across multiple domains, whereas the remain public specifications were obtained directly from their respective websites. The complete list of specifications can be seen in Appendix B.

Table 3.1: Distribution of the 50 OpenAPI Specifications used in this dissertation, showing the number of specifications per company and their respective domain.

Company	Domain	Number of Specifications
Adyen	Payments	3
Amadeus	Travel	6
Brex	Financial Services	4
Countrylayer	Geographic Data	1
DHL	Logistics	2
Youtube	Media Platform	2
Kong	API Infrastructure	11
Kuehne+Nagel	Logistics	21

3.3 Architecture

Figure 3.2 depicts the architecture of NysusAI. The input, provided by the user, is composed of the following:

- **OpenAPI Specification**: The OpenAPI Specification to be analyzed;
- **Output directory**: The directory where the output, a report with all the guidelines violations and API inconsistencies, will be stored;
- **GitHub Copilot API key**: The key to access and establish a communication with GitHub Copilot[12];

[6] https://countrylayer.com/
[7] https://www.dhl.com/us-en/home.html
[8] https://www.youtube.com/
[9] https://konghq.com/
[10] https://www.kuehne-nagel.com/
[11] https://apis.guru/
[12] https://docs.github.com/en/copilot

- **GitHub Copilot configuration**: The number of seconds the NysusAI will wait for the model's response before timing out, and the maximum number of retry attempts in case of a timeout;
- **AI model**: The name of the model that is going to analyze the OpenAPI Specification.

NysusAI is composed of five modules: the *Scraper Module*, which is responsible for scraping the Zalando RESTful API guidelines, processing their content, and storing them in the database; the *Orchestrator Module*, which is responsible for initializing all the components and for coordinating the validation pipeline. It establishes the connection to the database to fetch the Zalando guidelines, and connects and authenticates to the GitHub Copilot API; the *Validator Module*, which is responsible for validating the API specification against the guidelines, detecting internal inconsistencies, and deduplicating the issues flagged by the LLM; the *LLM Module*, which is responsible for managing the communication with the external API; and the *Report Module*, responsible for processing all the inconsistencies and guidelines violations, and generating a report with all the processed information.

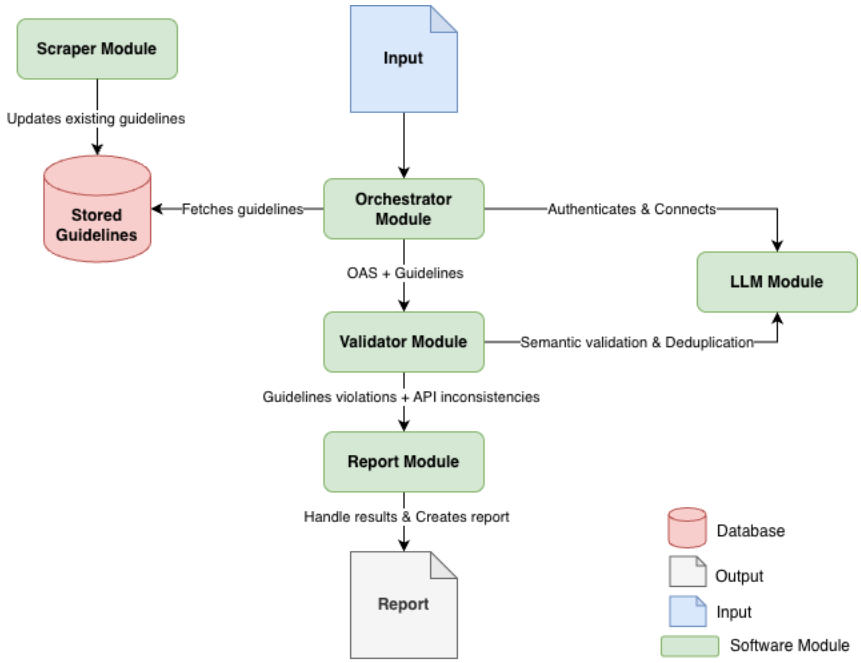


Figure 3.2: Architecture of NysusAI, illustrating the five modules and their interaction, from guideline retrieval and storage, to validation and report generation.

3.3.1 Scraper Module

The *Scraper Module* is responsible for automatically scraping and storing the Zalando guidelines, ensuring the most up-to-date version of the guidelines is always available. Figure 3.3 shows an overview of the scraper module. A web scraper extracts the HTML content of the Zalando web

page. The HTML is then parsed using Beautiful Soup[13], where only the guidelines are retrieved. To ensure that the guidelines are always updated, a job scheduler using GitHub Actions[14] is set to run the scraper once every 24 hours.

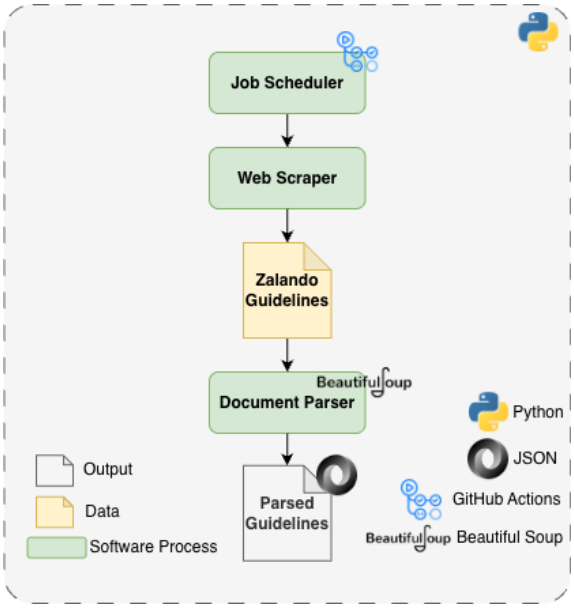


Figure 3.3: Data flow of the Scraper Module, depicting the pipeline from the job scheduler triggering the web scraper, through the parsing of the guidelines, to their storage.

Zalando's guidelines are composed of 15 sections, each containing a set of related guidelines, which are intended for the design of RESTful APIs and the OpenAPI Specification: (1) *General Guidelines*; (2) *REST Basics - Meta Information*; (3) *REST Basics - Security*; (4) *REST Basics - Data formats*; (5) *REST Basics - URLs*; (6) *REST Basics - JSON payload*; (7) *REST Basics - HTTP requests*; (8) *REST Basics - HTTP status codes*; (9) *REST Basics - HTTP headers*; (10) *REST Design - Hypermedia*; (11) *REST Design - Performance*; (12) *REST Design - Pagination*; (13) *REST Design - Compatibility*; (14) *REST Design - Deprecation*; and (15) *REST Operation*. Of these 15 sections, sections 1, 13, 14, and 15 were excluded because their guidelines are not relevant to the validation of the OpenAPI Specification. These sections mainly contain operational and organizational recommendations, such as "Must obtain approval of client before API shut down [185]", "MUST collect external partner consent on deprecation time span [186]", "SHOULD monitor API usage [193]", and "SHOULD provide API user manual [102]".

Using Beautiful Soup, each guideline was then scraped and transformed into a JSON object with the following fields:

- **Topic**: The ID of the section where the guideline is defined.

[13] https://www.crummy.com/software/BeautifulSoup/
[14] https://github.com/features/actions

- **Link**: The URL to the Zalando guidelines website with an anchor referencing the specific guideline. The anchor is embedded in the guideline title and enables direct navigation to the corresponding guideline within the page.
- **Title**: The title of the guideline.
- **Severity**: The severity of the guideline. The severity of each guideline can be found in the beginning of the guideline title in bold and takes one of three values: MUST, SHOULD, or MAY. These values are then mapped to ERROR, WARNING, and INFO, respectively.
- **Text**: The description of the guideline.

Listing 1 shows the JSON object representing the guideline shown in Figure 3.1

```
{
  "topic": "http-status-codes-and-errors",
  "link": "https://opensource.zalando.com/restful-api-guidelines/#243",
  "title": "Must use official http status codes",
  "severity": "ERROR",
  "text": "You must only use official HTTP status codes [truncated]"
}
```

Listing 1: JSON object representation of the Zalando guideline shown in Figure 3.1, containing the topic, link, title, severity, and description of the guideline.

In addition to the textual content, structured elements embedded within the guideline description were also parsed and extracted. Lists are extracted by detecting HTML `div` elements with one of the following classes: *ulist*, *dlist*, or *olist*. Each list is then parsed by extracting its full text content, which is applied to the *text* field followed by a double newline character to preserve separation from the surrounding content.

Code snippets are extracted by parsing HTML `code` elements. For each code block, the corresponding language is parsed from the *data-lang* attribute of the `code` HTML element. The content is then extracted and formatted using a Markdown code block with the detected language. To improve consistency, a normalization step is applied to grammar-based formats such as BNF and Augmented Backus–Naur Form (ABNF), as well as HTTP, where excessive whitespace is removed while preserving structural meaning. Figure 3.4a shows an example of a code snippet and Listing 3.4b its resulting code block. The resulting code block is then embedded in the *text* field as a string with line breaks represented using the escaped newline character `\n`.

Tables are extracted by parsing HTML `table` structures and transforming them into a JSON object. Table headers (`th`) are mapped to the *columns* field, while each subsequent row (`tr`) is transformed into a dictionary by mapping each entry to its corresponding column name. Figure 3.5a shows an example of a Zalando guideline table, and Listing 3.5b shows the resulting JSON representation. The resulting JSON object is then embedded in the *text* field as a string value while preserving its original structure through serialization using escaped characters.

```
<application-hostname> ::= <application-id>.<organization-unit>.zalan.do
<application-id>       ::= [a-z][a-z0-9-]*  -- application identifier
<organization-id>      ::= [a-z][a-z0-9-]*  -- organization unit identifier, e.g. team
identifier
```

(a) Code snippet from a Zalando guideline, showing a BNF grammar definition for application hostnames.

````
```bnf
<application-hostname> ::= <application-id>.<organization-unit>.zalan.do
<application-id> ::= [a-z][a-z0-9-]* -- application identifier
<organization-id> ::= [a-z][a-z0-9-]* -- organization unit identifier,
 ↪  e.g. team identifier
```
````

(b) Resulting Markdown code block after parsing the Zalando guideline code snippet.

Figure 3.4: Transformation of a Zalando guideline code snippet (top) into a structured Markdown code block (bottom) after parsing by the *Scraper Module*.

3.3.2 Database

To store the Zalando guidelines, a database was used. Since these guidelines need to be constantly and automatically updated, a cloud-hosted database solution was chosen. Table 3.2 presents the solutions that were considered, as well as their respective details.

Table 3.2: Comparison of the cloud-hosted database solutions considered for storing the Zalando guidelines.

Name	Storage	Price
AWS RDS [38]	20 GB	Free*
Supabase [56]	500 MB	Free

* AWS Free Tier only gives $100 USD worth of credits and is only valid for 1 year.

Considering the size of the Zalando's guidelines, and the fact that the AWS RDS Free Tier is only available for the first year, the Supabase solution was selected, as it provides enough storage capacity, as well as a fully managed environment with minimal operational overhead and no financial overhead.

3.3.3 LLM Module

The *LLM Module* is the core component of NysusAI, responsible for all interactions with the LLM. LLMs were chosen as the core technology due to their ability to understand and reason about both natural language guidelines, as well as structured API specifications, enabling a context-aware validation that goes beyond the capabilities of current static rule-based approaches. Figure 3.6 shows a detailed version of the data flow of the module. Each prompt is composed of two parts:

Parameter Type	Comma-separated Values	Multiple Parameters
Header	`style: simple, explode: false`	not allowed (see RFC 9110 Section 5.3)
Query	`style: form, explode: false`	`style: form, explode: true`

(a) Table from a Zalando guideline, showing the supported styles and explode values for header and query parameters.

```
{
  "columns": ["Parameter Type", "Comma-separated Values", "Multiple
   ↪  Parameters"],
  "rows": [
    {
      "Parameter Type": "Header",
      "Comma-separated Values": "style: simple, explode: false",
      "Multiple Parameters": "not allowed (see RFC 9110 Section 5.3)"
    },
    {
      "Parameter Type": "Query",
      "Comma-separated Values": "style: form, explode: false",
      "Multiple Parameters": "style: form, explode: true"
    }
  ]
}
```

(b) Resulting JSON representation after parsing the Zalando guideline table.

Figure 3.5: Transformation of a Zalando guideline table (top) into a structured JSON representation (bottom) after parsing by the *Scraper Module*.

a system prompt, which defines the model's role, rules, and expected output format, and a user prompt, which provides the input data, such as the OpenAPI Specification. This prompt is then sent to the GitHub Copilot API. A retry mechanism was implemented in the event that an error occurs with the GitHub Copilot API, such as request timeouts or empty responses. The LLM response is then parsed by a response parser, which uses a regular expression to extract the JSON array from the raw text returned by the model.

3.3.3.1 LLM

To ease integration with the existing Kuehne+Nagel workflow, GitHub Copilot was selected as it is already adopted within the company. One of the features GitHub Copilot provides is access to a range of models from which we can choose based on the Copilot Plan we have subscribed. Each one of these models has a number of premium requests associated with it. These premium requests act as a cost to run the model, where higher-tier models use a higher number of premium requests. These premium requests are then deducted from the GitHub account's monthly quota, which depends on the Copilot plan and resets at the beginning of each month. In this dissertation,

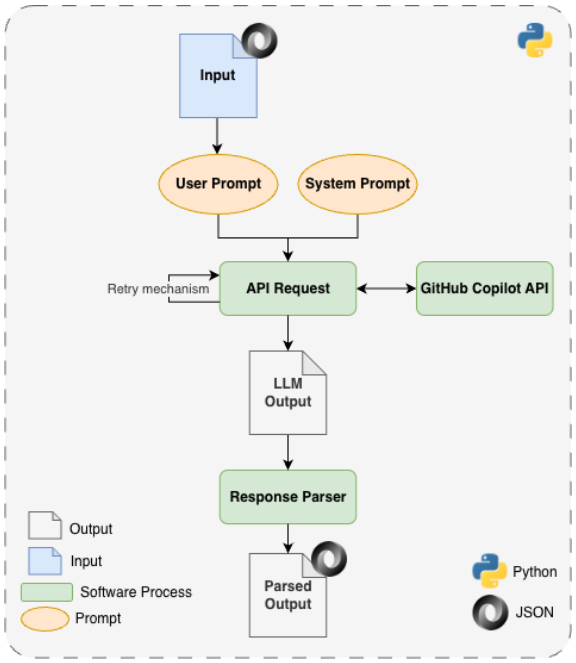


Figure 3.6: Data flow of the *LLM Module*, depicting the pipeline from the prompts through the API request to the GitHub Copilot API, to the parsing of the LLM output into a structured response.

a Copilot Student plan was used, which means a monthly quota of 300 premium requests [39]. The accessible models with their respective premium requests and input/output context windows can be seen in Table 3.3.

All the Codex models, Raptor Mini, and Grok Code Fast 1 were discarded because they are fine-tuned for software engineering applications, such as writing code and tests [109, 122, 124]. Gemini 2.5 Pro and Gemini 3 Pro were discarded because the models were set to be discontinued during the development of this dissertation [31]. Claude Haiku 4.5, Gemini 3 Flash, Gemini 3.1 Pro, GPT-5.1, and GPT-5.2 were also discarded due to the associated cost of running each model. Of the remaining models, GPT-4.1 and GPT-5 Mini, GPT-4.1 was selected. During initial trials, where both models were tested, GPT-5 Mini proved to have a slower execution time which resulted in a higher probability of receiving a session timeout. For this reason, GPT-4.1 was chosen.

Although GPT 4.1 supports a maximum input context window of 1M tokens, GitHub Copilot imposes a reduced limit of only 64k tokens, as shown in Figure 3.7.

```
(.venv) → zalando-tool git:(main) ✗ python3 -m asyncio
asyncio REPL 3.14.0 (main, Oct  7 2025, 09:34:52) [Clang 17.0.0 (clang-1700.3.19.1)] on darwin
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> from copilot import CopilotClient
>>> client = CopilotClient({"github_token":                                                    })
>>> await client.start()
>>> models = await client.list_models()
>>> gpt = next((model for model in models if model.id == "gpt-4.1"), None)
>>> gpt.capabilities.limits.max_prompt_tokens
64000
>>>
```

Figure 3.7: GitHub Copilot maximum prompt tokens for GPT-4.1, showing the 64k limited context window.

Table 3.3: Models accessible with the Copilot Student plan, showing their input and output token limits and premium request multipliers [40].

Model	Input tokens	Output tokens	Premium Requests
Claude Haiku 4.5 [10]	200k	64k	0.33
Gemini 2.5 Pro [29]	1M	64k	1
Gemini 3 Flash [32]	1M	64k	0.33
Gemini 3 Pro [33]	1M	64k	1
Gemini 3.1 Pro [30]	1M	64k	1
GPT-4.1 [100]	1M	32k	0
GPT-5 Mini [101]	400k	128k	0
GPT-5.1 [104]	400k	128k	1
GPT-5.1 Codex [103]	400k	128k	1
GPT-5.1 Codex Mini [102]	400k	128k	0.33
GPT-5.1 Codex Max [105]	400k	128k	1
GPT-5.2 [106]	400k	128k	1
GPT-5.2 Codex [107]	400k	128k	1
GPT-5.3 Codex [108]	400k	128k	1
Grok Code Fast 1 [124]	256k	10k	0.25
Raptor Mini [122]	264k	64k	0

3.3.3.2 Prompt Engineering

Prompt Engineering is very important when making prompts to LLMs. This is a set of techniques that influence the quality, relevance and accuracy of the LLM output by writing effective instructions to get the most accurate, helpful, and relevant outputs [27]. Three prompt engineering techniques were applied: Role Prompting, Few-Shot Prompting, and Structured Output Prompting, each addressing different aspects of the quality of the output:

Role Prompting Role Prompting assigns a persona to an LLM to guide the style and focus of its responses. By aligning the model's behavior with a specific role, the clarity and accuracy of the LLM output is improved, resulting in focused and contextually relevant answers [81, 85]. In this dissertation, this technique is used to assign the model the role of an API specification validator, consistency analyzer, and issue duplicate detector to improve the consistency and reliability of the generated findings, particularly when validating the specification against the Zalando guidelines or identifying internal inconsistencies. Listing 2 shows an example of the Role Prompting technique applied to the guidelines validation prompt.

```
You are an API Specification validator. You will receive excerpts of an API Specification and a set of guidelines rules.

Your job is to identify violations of the provided rules in the provided API Specification.
```

Listing 2: Role Prompting example from the guidelines validation prompt, showing how a persona is assigned to the LLM to guide its behavior during the validation process.

Few-Shot Prompting In Few-Shot Prompting, a small number of examples are introduced to the prompt so the LLM does not rely solely on its pre-existing knowledge. By providing these examples, the expected tone and level of detail are presented, helping the model to understand the context and expected output [83, 84]. This technique was applied to all the prompts, where examples of types of inconsistencies were provided to guide the LLM towards the types of inconsistencies it should look for, as well as examples of JSON paths, so every issue's path follows the same format. Listing 3 shows an example of the Few-Shot Prompting technique applied to the guidelines validation prompt.

```
"violation_path": {
  "type": "string",
  "description": "The JSON path in the Specification where the violation was found. Always use the JSONPath format (example: $.paths['/users'].post.responses['200'].content['application/json'].schema.$ref)."
},
```

Listing 3: Few-Shot Prompting example from the guidelines validation prompt, showing how a JSON path example is provided to guide the LLM towards the expected output format.

Structured Output Prompting Structured Output Prompting is a technique where the prompt defines the format and structure the LLM output must follow [86]. In order to parse the LLM response, this technique was used so the output is predictable and consistent. The prompts for guideline validation and inconsistency detection include a JSON schema specifying the information the LLM must return for each flagged issue. The schema defines the expected fields to return, as well as their types, and a description of what the field is supposed to be. Listing 4 shows the JSON schema used for the inconsistency detection prompt. For both the guidelines violation and inconsistency detection, the schemas share a common set of fields: a *violation_path*, which represents the path in the OpenAPI Specification where the issue was detected; a *reasoning*, explaining why the issue was flagged; a *severity*, indicating the severity of the issue which, for the guidelines violations, is the one of the

guidelines being violated, whereas for the inconsistency detection, it is inferred by the LLM based on the nature of the issue; and a *confidence* field, which represents a number between 0 and 100, representing how certain the LLM is about the flagged issue. The guidelines violation schema also includes a *title* field and a *link* field, containing the title and the link of the violated guideline, respectively. Regarding the deduplication prompt, a simpler JSON schema was defined since the LLM only needs to return a JSON array of the duplicate ids.

```
{
  "violation_path": {
      "type": "string",
      "description": "The JSON path in the Specification where
       ↪ the violation was found. Always use the JSONPath
       ↪ format (example:
       ↪ $.paths['/users'].post.responses['200']
       ↪ .content['application/json'].schema.$ref)."
  },
  "reasoning": {
    "type": "string",
    "description": "A brief explanation of why the excerpt of
     ↪ the specification is inconsistent."
  },
  "severity": {
    "type": "string",
    "enum": ["ERROR", "WARNING", "INFO"],
    "description": "The severity level of the inconsistency."
  },
  "confidence": {
    "type": "number",
    "description": "A score between 0 and 100 indicating how
     ↪ certain you are that this is a real inconsistency. 100
     ↪ means absolutely certain. 0 means a guess."
  }
}
```

Listing 4: Structured output JSON schema for inconsistency detection, defining the expected fields, their types, and descriptions that the LLM must return for each flagged inconsistency.

The complete prompts for guidelines validation, inconsistency detection, and deduplication can be found in Appendix C.

3.3.4 Validator Module

The *Validator Module* is responsible for preparing the information to be sent to the LLM, coordinating the LLM calls, and post-processing its results, as shown in Figure 3.8. Since the objective is to validate an API specification, instead of relying on a single and generic prompt, a prompt engineering technique called Prompt Chaining was used. Prompt Chaining breaks complex tasks

into smaller ones, allowing each prompt to be designed for a specific task, obtaining more accurate and relevant responses [58]. Three prompts were created, each one with a specific task. The first prompt, referred to as guidelines validation prompt, is designed to validate the specification against the Zalando guidelines; the second one, referred to as inconsistency detection prompt, is responsible for detecting inconsistencies inside the API; the third one, referred to as deduplication prompt, is customized to deduplicate the issues found. These three prompts end up forming a pipeline where the output of the first two prompts are used as input to the third prompt.

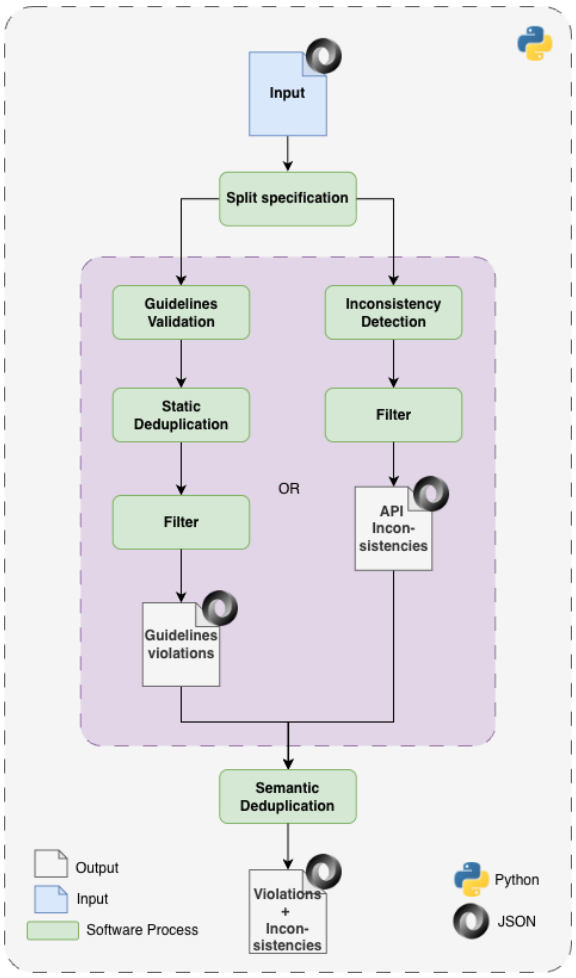


Figure 3.8: Data flow of the *Validator Module*, depicting both paths for guideline validation and inconsistency detection, followed by the semantic deduplication to produce the final list of violations and inconsistencies.

3.3.4.1 Specification and Guidelines Excerpts

One of the problems current AI models have is the limited size of their input context window, which limits the amount of information we can provide to the LLM. In addition, the bigger the input, the less capacity the model has to retain the context of all of it, a phenomenon called Lost in the Middle [17].

To tackle these problems, the API specification is divided into multiple excerpts to reduce its size, which are used for both guidelines violations and inconsistency detection prompts. Each excerpt contains only one endpoint from the original specification and includes all the required *components* definitions. These definitions are initially obtained by extracting the *$ref* references from the endpoint, and then recursively traversing these references to identify all the definitions on which they depend, either directly or indirectly, ensuring that the excerpt only contains the *components* definitions that are required by the endpoint. In addition, each excerpt also includes the necessary metadata to process the endpoint correctly, such as the *security* and *servers* fields, which defines the authentication requirements and the base URL of the API.

For the guidelines validation prompt, an additional excerpt containing only the metadata of the specification is created. This excerpt includes information that is not specific to individual endpoints, such as the *version* and the *info* fields, which contains information about the version of the API, as well as the contact information and the identifier of the specification. The metadata excerpt is validated against the *Meta Information* section of the Zalando guidelines, while the endpoint excerpts are validated against the remaining sections. This ensures that each excerpt is only validated against the guidelines relevant to its content, reducing the size of each prompt.

For the inconsistency detection prompt, an additional excerpt is created to detect API inconsistencies among the endpoints. This excerpt contains a small version of every endpoint, where each only has their parameters, request bodies, and responses, as well as the complete *components* section, allowing the LLM to identify cross-endpoint inconsistencies, such as duplicated inline requests, response structures that should be extracted into reusable schema components, inconsistent parameter names, broken *$ref* references and missing schema definitions.

Table 3.4 shows the average token usage ratio across the tested API specifications by company, where the ratio represents the average number of tokens of the excerpts as a percentage of the total number of tokens of the original versions. The *Specification* column baseline is the original specification, whereas for the *Guidelines Prompt* and *Inconsistencies Prompt* columns, the baseline is the full specification combined with all the guidelines. The ratio for every API can be seen in Appendix D. All of these tokens were obtained using the Tiktoken[15] library, with an encoder configured for the `gpt-4.1` model.

3.3.4.2 Deduplication and Filtering

After obtaining the violations flagged by the LLM, a static deduplication step is applied only to the guidelines violations. This step removes any violation that shares the same *violation_path* and *link* as an already seen violation, as it is a duplicated violation. This step cannot be performed on the inconsistencies found by the LLM because they do not have a field that uniquely identifies the issue flagged, such as the *link* field of the guidelines violation schema. A confidence filter is then applied to each flagged issue to remove low-confidence issues. A threshold of 70 was implemented, where every issue with confidence below 70 was removed, as sometimes the LLM

[15] https://github.com/openai/tiktoken

Table 3.4: Average token usage ratio across API specifications by company, where the Specification column shows the average size of each excerpt relative to the full specification, the Guidelines Prompt column shows the size of each excerpt combined with its selected guidelines relative to the full specification combined with all guidelines, and the Inconsistencies Prompt column shows the size of each excerpt relative to the full specification used in the inconsistency prompt.

API	**Average Ratio**		
	Specification	Guidelines Prompt	Inconsistencies Prompt
Adyen	45%	64%	35%
Amadeus	61%	54%	47%
Brex	39%	70%	32%
Countrylayer	18%	52%	11%
DHL	28%	60%	21%
Youtube	35%	68%	31%
Kong	42%	67%	35%
Kuehne+Nagel	48%	56%	37%

would hallucinate, flagging issues that do not represent a guideline violation, as shown in Figure 3.9.

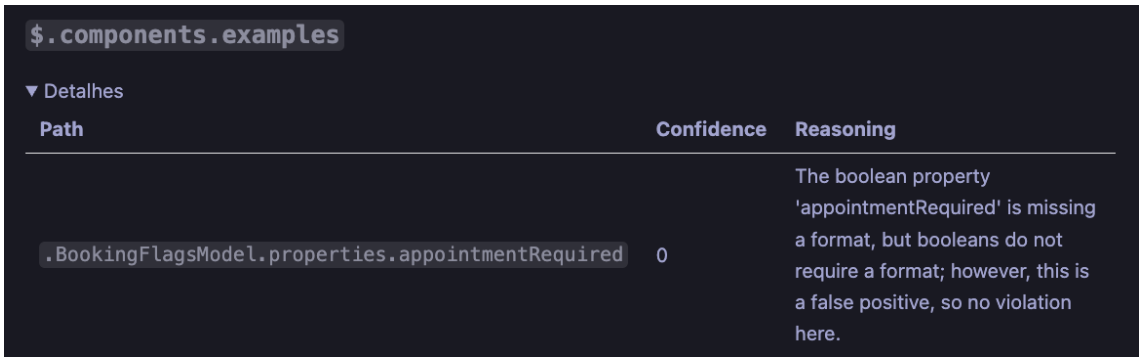


Figure 3.9: Example of a low-confidence false positive issue flagged by the LLM, where the model assigns a confidence of zero and acknowledges that the flagged issue is a false positive.

After both guidelines violations and API inconsistencies pass through the filtering phase, they go into a semantic deduplication phase, where all the remaining issues are sent to the LLM to be deduplicated. A static approach is not enough because two issues may describe the same underlying problem using different *violation_path* fields or different wording in their *reasoning* field, making exact field matching ineffective. By leveraging the LLM semantic understanding, issues that are conceptually identical but textually different can be identified and removed.

3.3.5 Report Module

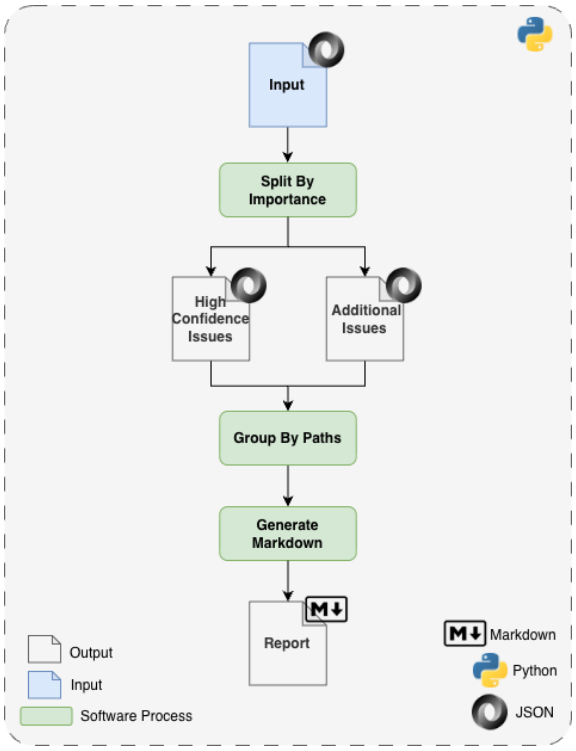


Figure 3.10: Data flow of the *Report Module*, illustrating how issues are processed to generate the final Markdown report.

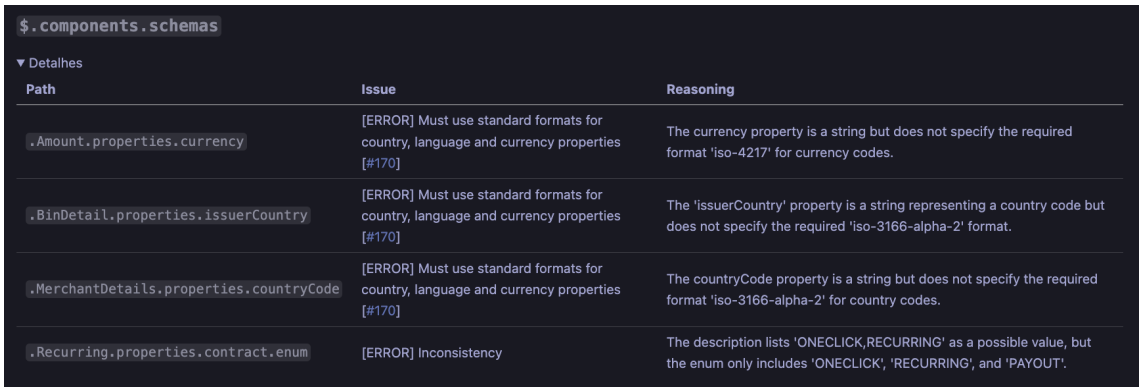
After all issues have been deduplicated, they are passed to the *Report Module*, which is responsible for processing them and generating a Markdown report, as illustrated in Figure 3.10. Given that LLMs can flag a large number of issues, the report is divided into two sections to prioritize the review of the critical issues while still including the lower-confidence issues: *High-Confidence Issues* and *Additional Issues*. *High-Confidence Issues* contain the most critical issues. For guideline violations, an issue is classified as critical if its *severity* is *ERROR* and its *confidence* level is 100. For inconsistencies, only the *severity* is required to be *ERROR*, as it was observed that the LLM rarely assigns a high *confidence* number to inconsistencies, which would result in no inconsistencies being classified as critical. The remaining issues are placed under the *Additional Issues* section. The issues within each section are then grouped by the common sub-path of their *violation_path*, ensuring that issues belonging to the same location are presented together for improved readability. Within each group, issues are then sorted first by their *violation_path* and then by their *severity* (*ERROR*, *WARNING*, *INFO*) to help the user focus on the most important issues first within each group. For each group, a Markdown table is created where each issue is presented, giving information about the following details:

Path The specific part of the *violation_path* of the issue.

Issue The severity of the flagged issue. For guidelines violations, the title of the violated guideline is also presented, along with a hyperlink to the corresponding guideline on the Zalando website, represented as the identifier of the violated guideline. For inconsistencies, the label *Inconsistency* is added instead.

Reasoning The explanation of why the issue was flagged.

The report also includes the `<details>` HTML element to allow the user to minimize both entire groups and sections, making it easier to navigate the report when the number of issues is very large. An example of a generated Markdown table can be seen in Figure 3.11. A complete generated report can be seen in Appendix E.



`$.components.schemas`

▼ Detalhes

Path	Issue	Reasoning
`.Amount.properties.currency`	[ERROR] Must use standard formats for country, language and currency properties [#170]	The currency property is a string but does not specify the required format 'iso-4217' for currency codes.
`.BinDetail.properties.issuerCountry`	[ERROR] Must use standard formats for country, language and currency properties [#170]	The 'issuerCountry' property is a string representing a country code but does not specify the required 'iso-3166-alpha-2' format.
`.MerchantDetails.properties.countryCode`	[ERROR] Must use standard formats for country, language and currency properties [#170]	The countryCode property is a string but does not specify the required format 'iso-3166-alpha-2' for country codes.
`.Recurring.properties.contract.enum`	[ERROR] Inconsistency	The description lists 'ONECLICK,RECURRING' as a possible value, but the enum only includes 'ONECLICK', 'RECURRING', and 'PAYOUT'.

Figure 3.11: Example of issues grouped under the *$.components.schemas* sub-path, showing guidelines violations and an inconsistency with their respective paths, issues, and reasoning.

3.3.6 NysusAI Interface

NysusAI can be used in two different ways: (1) via Graphical User Interface (GUI) and (2) via Command Line Interface (CLI). By default, NysusAI launches in the GUI mode, where a window will be created so the user can enter the necessary information to run it, as shown in Figure 3.12.

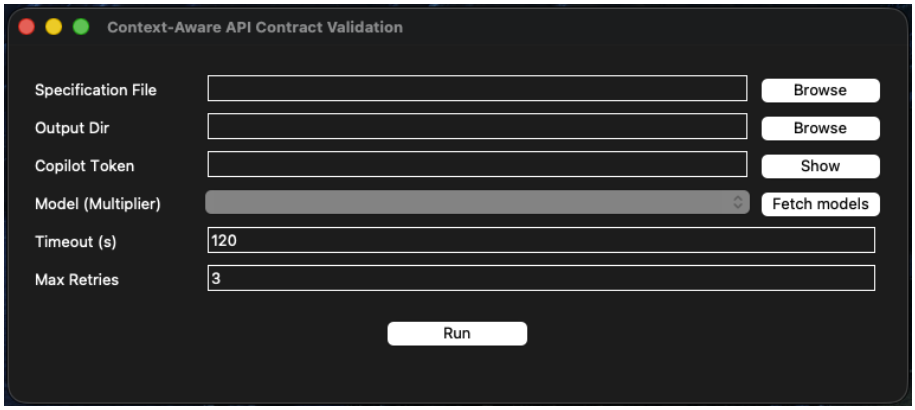


Figure 3.12: NysusAI GUI, showing the input fields the user needs to fill to run the tool.

For both the *Specification File* and *Output Dir* fields, when the user clicks on the "Browse" button, a file browser will open so the user can select both the OpenAPI Specification file and the directory where the report will be stored, respectively. For the *Copilot Token* field, the user needs to insert the token associated with their GitHub account. When the user inserts the token, it will be masked with the $\star$ character, which can be revealed by clicking on the "Show" button. Then, the user needs to click on the "Fetch models" button so NysusAI can fetch all the models the token has access to. These models will be displayed on a dropdown list on the *Model (Multiplier)* field, along with the premium requests associated to run each model. The last two fields are related to the LLM settings. The *Timeout (s)* field is the number of seconds NysusAI will wait for the LLM response, with a default value of 120 seconds, whereas the *Max Retries* field defines the number of retry attempts the retry mechanism will try to obtain a response from the LLM, in the event of a timeout or an empty response. After running NysusAI, a JSON file named *config.json* is created in the same folder where the NysusAI is with the information provided by the user in the GUI. This file is used to pre-fill the GUI on future runs, so the user does not need to re-enter the information every time. This file can also be used to run NysusAI via CLI, by providing the `-c` or `--cli` flag on the command line. This will cause NysusAI to run without showing the user interface, obtaining the necessary information to run by the *config.json* file. The *config.json* has the following fields:

- *spec_path*: The absolute path of the OpenAPI Specification that is going to be analyzed.
- *output_dir*: The absolute path of the directory where the report file will be stored.
- *copilot_token*: The token for Copilot API access.
- *model*: The name of the model that will analyze the OpenAPI Specification.
- *timeout*: The number of seconds to wait for the model response.
- *max_retries*: The number of retry attempts on timeout.

3.4 Summary

This chapter presented the architecture and the details of NysusAI, as well as the desired functionalities and the guidelines used. Since Kuehne+Nagel guidelines are only available internally, the Zalando guidelines were chosen as a replacement, as they are publicly available and represent a comprehensive and well-established set of industry-grade guidelines, covering a wide range of topics. A job scheduler is responsible for running a web scraper once every 24 hours to scrape and store the guidelines in a cloud-hosted database, which is crucial because the guidelines need to be constantly updated. The core component of NysusAI is the usage of an LLM which can process natural language to detect and deduplicate issues that current static-based tools cannot detect. Since current LLMs are quite limited in the input context window, and since they cannot hold the context of a lot of text, a technique was used where the OpenAPI Specification is split

into multiple smaller excerpts and where each excerpt is only validated against the relevant parts of the guidelines, helping to reduce the size of the prompts. The use of Prompt Engineering was also essential to improve the quality of the LLM responses by providing a persona to the LLM, examples, defining a structured output format, and decomposing the validation task into multiple specialized prompts instead of relying on a single generic one. This output is then used to create a report in which the issues are presented to the user.

The combination of a cloud-hosted database, an automated scraper that keeps the guidelines up to date, and a context-aware validation pipeline makes NysusAI suitable for use in industrial environments, where guidelines evolve over time and validation must scale across large and complex API specifications, a capability that current static validation tools and context tools, such as LintGPT, do not offer.

Chapter 4

Validation

In this chapter the experiments used to validate and evaluate NysusAI are presented. For each experiment, the setup, results and conclusions are presented.

Section 4.1 presents the first experiment, focused on evaluating the reliability of NysusAI's output. Section 4.2 presents the second experiment, focused on measuring the impact NysusAI has on its users. All of the research questions and the hypothesis were responded in Section 4.3. Section 4.4 identifies the main threats to the validity of the results. The limitations of NysusAI are presented in Section 4.5. Section 4.6 presents a summary of what was done in this chapter.

4.1 Output Reliability Evaluation

To measure the reliability of NysusAI's output, an experiment was performed in which NysusAI was used to validate a set of OpenAPI Specifications, where the flagged issues were reviewed, and the false discovery rate was calculated.

4.1.1 Experimental Setup

To measure NysusAI's reliability, all 50 OpenAPI Specifications from the dataset described in Section 3.2.2 were validated using NysusAI. All validations were performed using the `gpt-4.1` model. Since NysusAI relies on an LLM to find the guidelines violations and API inconsistencies, NysusAI's output is non-deterministic. For this reason, each OpenAPI Specification was validated a total of three times. If an issue was flagged by NysusAI on at least two runs, the issue was considered for this experiment, while the issues that appeared only on one run were considered outliers and discarded. Each considered issue was then manually reviewed and classified as true positive or false positive. Since this experiment uses API specifications from real-world companies, the total number of inconsistencies and guidelines violations are unknown, making it impossible to compute standard statistical metrics, such as recall and accuracy. For this reason, the evaluation of NysusAI focuses on its false discovery rate, which measures the proportion of flagged issues that are false positives, which can be calculated by the following equation:

$$FDR = \frac{FP}{FP + TP} \tag{4.1}$$

where FDR is the false discovery rate, FP is the number of false positives, and TP is the number of true positives.

4.1.2 Experimental Results

Figure 4.1 presents the classification of all issues per company, showing the number of true positives, false positives, and false discovery rate for each company. NysusAI detected a total of 1212 issues across all specifications, with 743 being guidelines violations and 469 API inconsistencies. Overall, NysusAI achieved a false discovery rate of 18.6%, indicating that approximately 4 out of 5 flagged issues were classified as true positives. Notably, this overall rate masks a significant difference between the two categories of issues, with guidelines violations having a significantly lower false discovery rate than API inconsistencies, as will be discussed in the following sections.

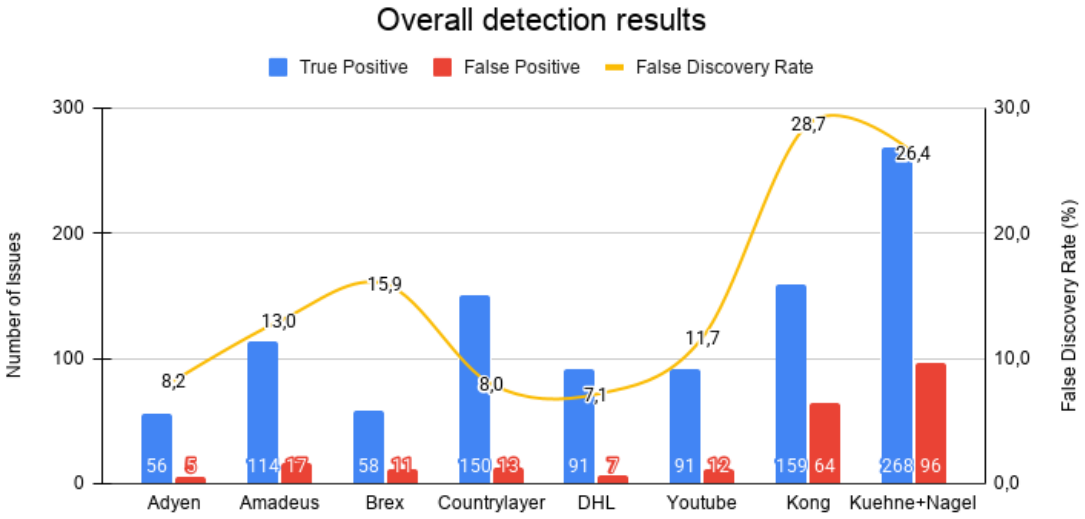


Figure 4.1: Overall issue detection results per company, showing the number of true positives, false positives, and the false discovery rate for each company.

4.1.2.1 Guidelines Violations Detection Results

Figure 4.2 shows the distribution of the number of true positives, false positives, and false discovery rate per company for guidelines violations. The results show that, compared to the overall results, the number of false positives is significantly lower and, consequently, so is the false discovery rate. This can be attributed to the high level of detail the context given to the LLM has, which in this case are the Zalando guidelines. In this case, the LLM has a set of rules which serve as an explicit reference on how the OpenAPI Specification should be, containing detailed rule descriptions, tables with the supported values, and code snippets with concrete examples. This

reduces the ambiguity faced by the LLM, as it only needs to compare the specification against a well-defined and detailed guideline to determine whether a guideline violation exists or not.

One example is Countrylayer, where all of the 91 issues were flagged as true positive. This can be explained by the nature of the identified violations, where the majority (82%) are related to special properties, such as country, country codes, and currencies, whose formats are explicitly defined in the guidelines, which reduces the likelihood of flagging false positive issues.

In contrast, Kuehne+Nagel's specifications have the highest false discovery rate. A significant proportion of the false positive issues are related to an OAuth2 schema implicit flow, which, according to the guidelines, this security feature should only be used when a full OAuth2 authorization flow is fully supported. However, this is not possible to verify based on the API specification alone. For this reason, all the flagged issues were marked as false positives.

Overall, NysusAI managed to achieve a false discovery rate of 8.5% due to the detail of the provided context, with six out of eight companies achieving a rate below 10%, demonstrating that providing context in a structured format contributes positively to the precision of NysusAI.

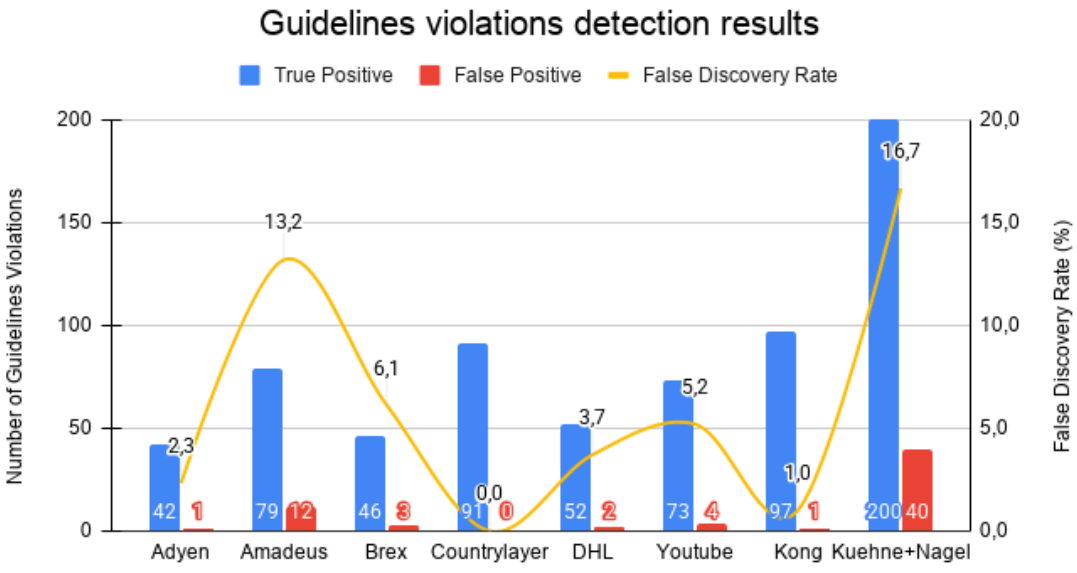


Figure 4.2: Overall guidelines violations detection results per company, showing the number of true positives, false positives, and the false discovery rate for each company.

Of the 680 guidelines violations classified as true positives, 547 (80%) can be detected using static analysis tools, while 133 (20%) can only be detected using context-aware tools. This discrepancy is expected, as static guidelines are prone to be more frequently violated since these are composed by guidelines that are supposed to be applied across all properties and endpoints, such as data formats and naming conventions, as shown in Figure 4.3.

Regarding the guidelines that can only be validated with context-aware tools, Figure 4.4 shows that guideline number 105 is the most frequently violated, followed by the guideline number 104, both related to security issues. The remaining guidelines cover a diverse range of topics, such

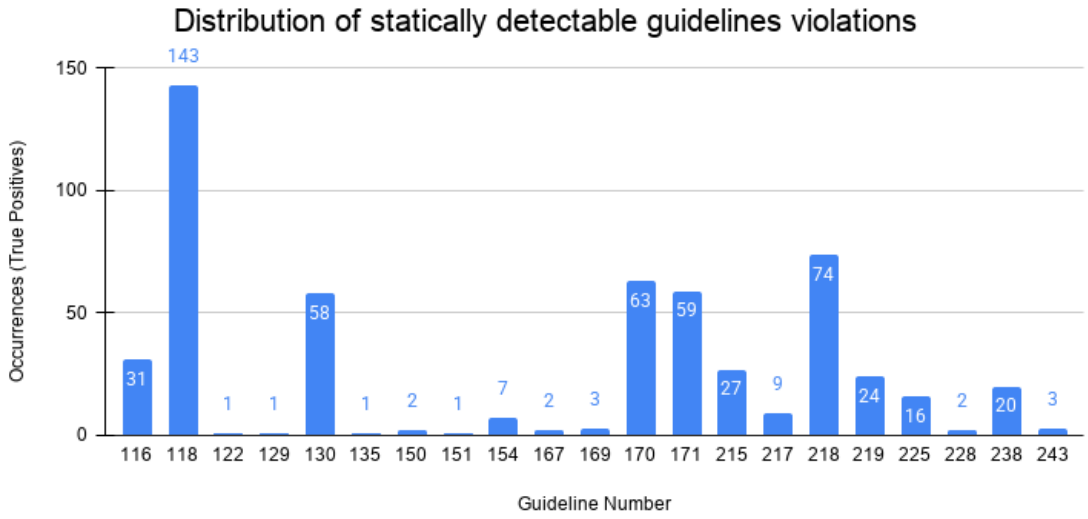


Figure 4.3: Distribution of the true positive guideline violations detectable by static tools, showing the number of occurrences per guideline number.
Note: #116: MUST use semantic versioning; #118: MUST property names must be snake_case (and never camelCase); #122: MUST not use null for boolean properties; #129: MUST use kebab-case for path segments; #130: MUST use snake_case (never camelCase) for query parameters; #135: SHOULD not use /api as base path; #150: SHOULD only use most common HTTP status codes; #151: MUST specify success and error responses; #154: MUST define collection format of header and query parameters; #167: MUST use JSON as payload data interchange format; #169: MUST use standard formats for date and time properties; #170: MUST use standard formats for country, language and currency properties; #171: MUST define a format for number and integer types; #215: MUST provide API identifiers; #217: MUST use full, absolute URI for resource identification (with same host, if applicable); #218: MUST contain API meta information; #219: MUST provide API audience; #225: MUST follow the naming convention for permissions (scopes); #228: MUST use URL-friendly resource identifiers; #238: MUST use standard data formats; #243: MUST use official HTTP status codes;

as URL design, HTTP methods and status codes, pagination, and JSON payload semantics. Despite the significant difference in the total number of violations, both static and context-dependent guidelines show a similar number of guidelines flagged, with 21 for static guidelines and 16 for context-dependent guidelines. This demonstrates that NysusAI provides broad coverage across both validation categories, being capable of identifying violations that static analysis tools cannot detect.

Figure 4.5 shows the distribution of the guidelines violations classified as false positives per guideline. The majority of false positives belong to the *MUST secure endpoints* guideline. As discussed above, this is due to the fact that the LLM does not have enough information to verify whether the specification follows the guideline or not, which resulted in a lot of false positives due to the size of the Kuehne+Nagel's specifications. The remaining false positives are distributed among the other guidelines, with *MUST property names must be snake_case (and never camelCase)* and *MUST use snake_case (never camelCase) for query parameters* guidelines showing a slightly increase, as the LLM is prone to flag single words, such as *name*, as not respecting these naming conventions, resulting in false positives.

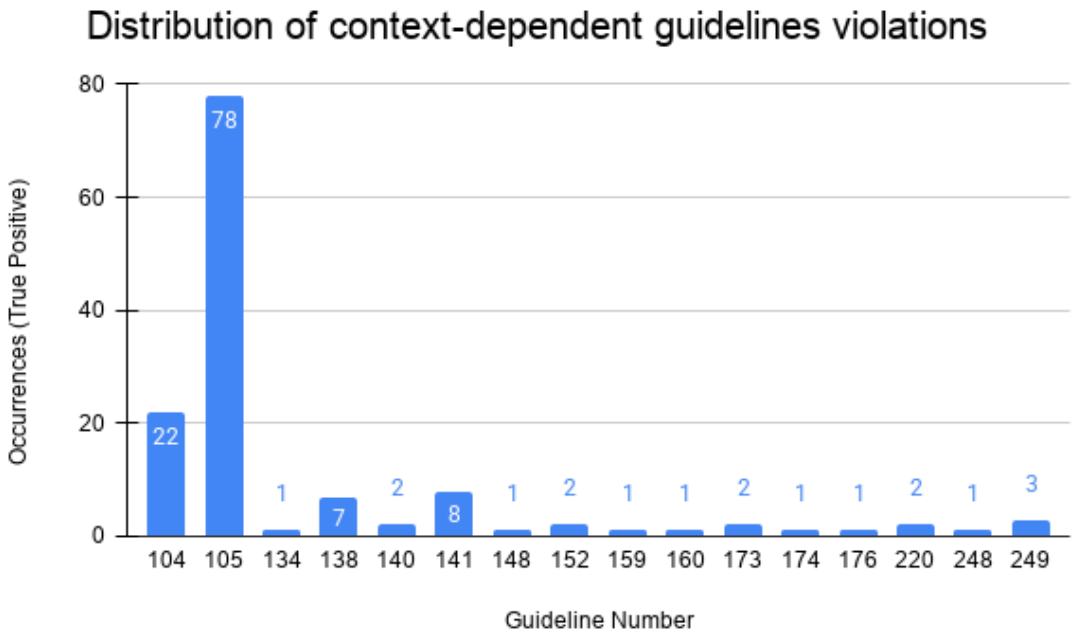


Figure 4.4: Distribution of the true positive guideline violations requiring context-aware tools, showing the number of occurrences per guideline number.
Note: #104: MUST secure endpoints; #105: MUST define and assign permissions (scopes); #134: MUST pluralize resource names; #138: MUST avoid actions — think about resources; #140: SHOULD define useful resources; #141: MUST keep URLs verb-free; #148: MUST use HTTP methods correctly #152: MUST use code 207 for batch or bulk requests; #159: MUST support pagination; #160: SHOULD prefer cursor-based pagination, avoid offset-based pagination; #173: MUST use the common money object; #174: MUST use common field names and semantics; #176: MUST support problem JSON; #220: MUST use most specific HTTP status codes; #248: SHOULD use pagination response page object; #249: MUST use the common address fields;

4.1.2.2 API Inconsistency Detection Results

Figure 4.6 shows the distribution of the number of true positives, false positives, and false discovery rate per company for API inconsistencies. Compared to both overall results and guidelines violations results, the false discovery rate for API inconsistency detection is significantly higher, ranging from 11.4% to 50.4%. This can be explained by the limited and small amount of context provided to the LLM. While in the guidelines violation prompt the context is composed by a detailed set of guidelines, in the inconsistency detection prompt the context provided to the LLM is only composed of a small set of types of API inconsistencies, leaving the LLM to reason only about the specification, making it more likely to flag false positive issues.

The high false discovery rate for Kong (50.4%), Kuehne+Nagel (45.2%), and Brex (40.0%) can be partially explained by the nature of their specifications, where companies have their own domain-specific conventions. Since the LLM lacks the necessary domain knowledge to distinguish intentional design decisions from real inconsistencies, the likelihood of flagging false positive issues is higher, leading to higher false discovery rates.

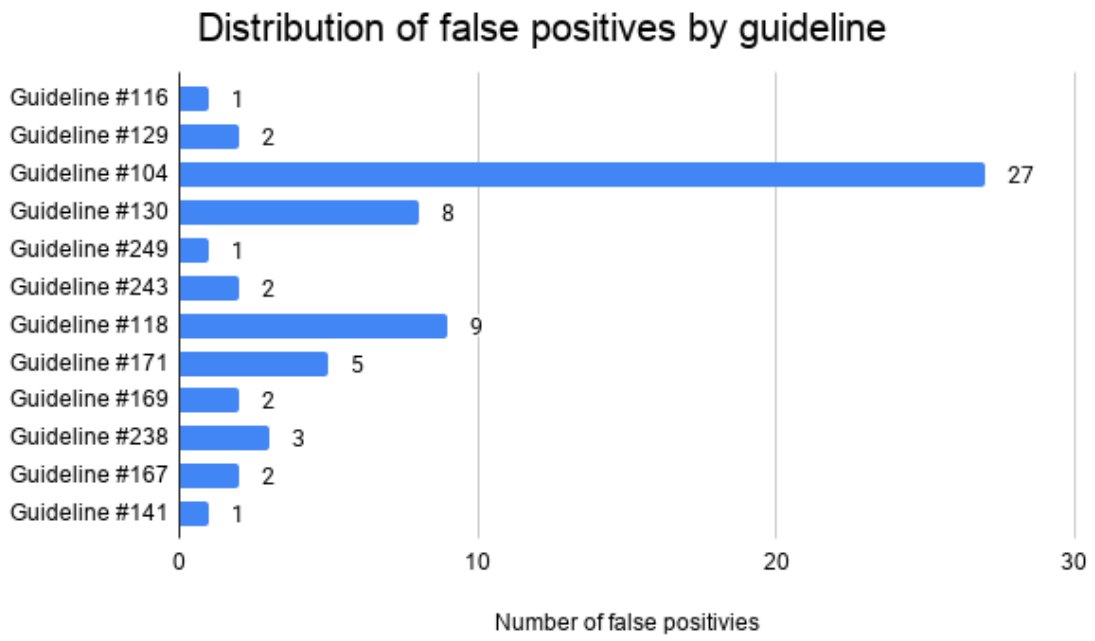


Figure 4.5: Distribution of false positive guideline violations, showing the number of occurrences per guideline number.
Note: #116: MUST use semantic versioning; #129: MUST use kebab-case for path segments; #104: MUST secure endpoints; #130: MUST use snake_case (never camelCase) for query parameters; #249: MUST use the common address fields; #243: MUST use official HTTP status codes; #118: MUST property names must be snake_case (and never camelCase); #171: MUST define a format for number and integer types; #169: MUST use standard formats for date and time properties; #238: MUST use standard data formats; #167: MUST use JSON as payload data interchange format; #141: MUST keep URLs verb-free;

In general, a false discovery rate of 34.5% was achieved in the detection of API inconsistencies, a value significantly higher than the false discovery rate of guidelines validation, due to the absence of a detailed context for the LLM to compare against.

To obtain a better perspective of the inconsistencies, all 307 inconsistencies classified as true positives were manually categorized into the five following categories:

- **Structural Issues**: Violations of the OpenAPI Specification structure, such as broken *$ref* references and objects without the *type: object* declaration.
- **Type and Schema Inconsistencies**: Mismatches between defined types and their actual usage, such as properties with incorrect types.
- **Example Inconsistencies**: Discrepancies between the examples and the schema definitions, such as an example value being a string, whereas its schema defines it as a number.
- **Reusability Issues**: Inline schemas duplicated across endpoints that can be extracted into a reusable component into the *components* section.
- **Misleading or Incorrect Descriptions**: Typos, misleading descriptions, properties names and names of the operations.

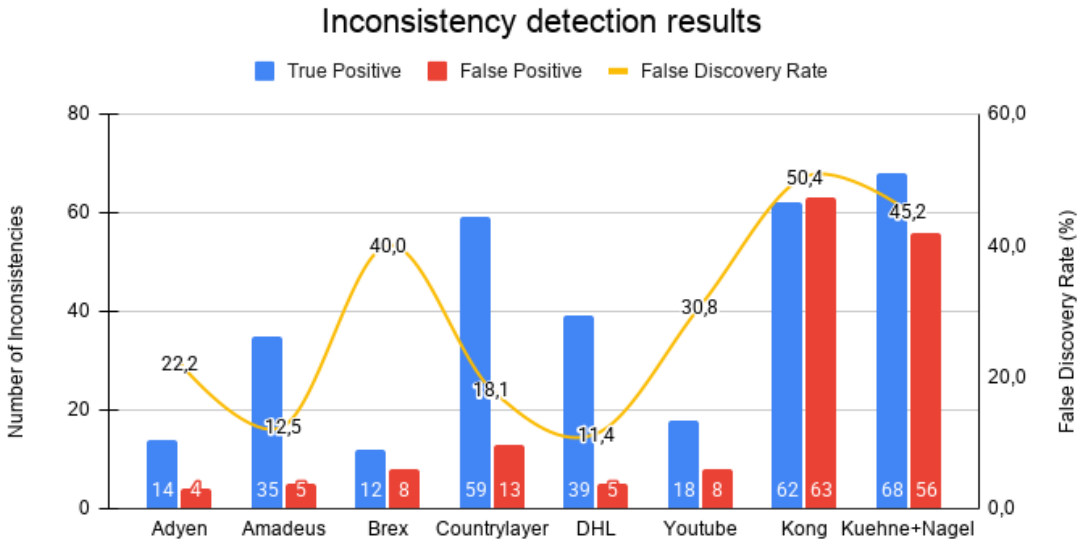


Figure 4.6: Inconsistency detection results per company, showing the number of true positives, false positives, and the false discovery rate for each company.

NysusAI managed to find inconsistencies similarly across all categories, with *Type and Schema Inconsistencies* being the category with the highest number of inconsistencies (78), as can be seen in Figure 4.7a. The high occurrence of this category is expected, given the large number of ways in which types, constraints, and schema definitions can diverge within a specification. *Misleading or Incorrect Descriptions* is the second most prevalent category with 62 inconsistencies. This result can be explained by the large amount of copied descriptions from other endpoints without updating them, ending up misleading the user, or by writing a word with a typo once and then copying it to other endpoints, without verifying if the word is spelled correctly. *Reusability Issues* are also frequent, with a total of 58 inconsistencies, suggesting developers tend to define inline schemas and parameters that are duplicated across endpoints, rather than extracting it into a reusable component. *Structural Issues* reflects issues that violate the OpenAPI Specification format and structure, which could cause parsing errors or unexpected behavior in tools used in API testing, with a total of 55 inconsistencies. *Example Inconsistencies*, with a total of 54 inconsistencies, reflect the discrepancies between the examples and the schema definitions, which may mislead developers when using the API.

Of these 307 inconsistencies, 133 inconsistencies could only be detected using a context-aware tool, which were categorized into the following seven categories:

- **Schema Mismatch**: The description of a property contradicts the schema definition, such as a description describing the endpoint returns an array of *Report Types*, but the endpoint returns an array of *Report Objects*.
- **Typo**: Spelling errors, such as a property named *longiture* instead of *longitude*.

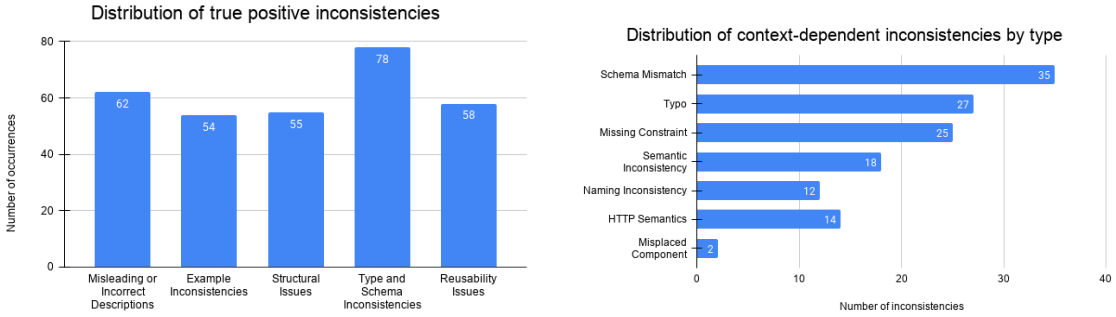
- **Missing Constraint**: A property lacks a constraint implied by its description, such as a description describing the only accepted values, but the property does not contain an *enum* field.
- **Semantic Inconsistency**: A property's type or name is inconsistent with its intended meaning, such as a boolean concept defined as a string.
- **Naming Inconsistency**: A title does not match its schema, or a property is named differently from similar concepts from other endpoints, such as *portOfDischargeLocation* and *portOfDelivery*.
- **HTTP Semantics**: HTTP status codes or methods are used in a way that breaks their semantics, such as an HTTP 200 status code described as an error.
- **Misplaced Component**: A component is defined in the wrong section of the specification, such as defining a response object where schemas are defined.

The distribution of the issues across these seven categories, as shown in Figure 4.7b, suggest that NysusAI is capable of detecting a broad range of issues that require reasoning about the intent and context of the API specification, which static analysis tools cannot identify. The fact that *Schema Mismatch*, *Typo*, *Missing Constraint*, *Semantic Inconsistency* and *Naming Inconsistency* compose 117 (88%) of the issues detected shows how poorly maintained real-world OpenAPI Specifications tend to be, with descriptions that contradict their schema definition, spelling errors, unenforced validation rules, properties with semantically incorrect types, and naming inconsistencies among endpoints. *HTTP Semantics* and *Misplaced Component* suggest that developers are also prone to both incorrectly structuring their OpenAPI Specifications, as well as describing their endpoint operations incorrectly.

Overall, these results demonstrate that NysusAI is capable of detecting a diverse range of API inconsistencies, ranging from issues that can be identified with static analysis tools, such as *Structural Issues* and *Reusability Issues*, as well as issues that require contextual reasoning, such as *Schema Mismatch*, *Missing Constraint*, and *Semantic Inconsistency*.

4.1.3 Experimental Conclusions

The results demonstrate that an LLM can effectively identify both guidelines violations and API inconsistencies in OpenAPI Specifications, with an overall false discovery rate of 18.6%. The importance of having a detailed and structured context was shown to be essential to obtain more reliable outputs. This impact can be seen in the guidelines violations results, which achieved a false discovery rate of just 8.5%, which is significantly lower than the 34.5% achieved by the inconsistencies detection. By providing detailed context, the LLM has a basis to compare the specification against, reducing ambiguity and improving its precision. On the other hand, without the context, the LLM must reason about the specification independently, increasing the likelihood of flagging false positive issues. Furthermore, NysusAI demonstrated the capacity to analyze a



(a) Distribution of the 307 true positive inconsistencies across the five categories.

(b) Distribution of the 133 context-dependent inconsistencies across the seven categories.

Figure 4.7: Distribution of the 307 true positive inconsistencies across five categories (left) and the 133 context-dependent inconsistencies across seven categories (right).

broad range of guidelines and inconsistencies. NysusAI detected violations across 21 guidelines that can be validated with static tools and 16 guidelines that require contextual reasoning. Regarding inconsistency detection, the 307 true positives inconsistencies were distributed across five categories, with 133 of them representing context-dependent inconsistencies that could only be detected by reasoning about the specification's context. These results confirm that NysusAI is capable of identifying a diverse range of issues that static analysis tools cannot detect.

4.2 User Study

To determine the impact NysusAI has on users, a user study was conducted. Participants were asked to validate OpenAPI Specifications against the Zalando guidelines, some manually, and some with the assistance of NysusAI. After completing the validation step, participants completed a survey designed to assess their experience with NysusAI.

4.2.1 Experimental Setup

Participants were asked to validate a total of six OpenAPI Specifications against the Zalando guidelines. The first three were validated manually, whereas the last three were validated with the assistance of NysusAI. After validating the specifications, each participant completed a survey, which is composed of two parts: (1) User Profiling and (2) Technology Acceptance Model (TAM) Evaluation. The first part, User Profiling, aims to assess the background of each participant, as well as their knowledge on the technologies used. The second part, TAM Evaluation, aims to verify the impact NysusAI has on its users by calculating the following metrics:

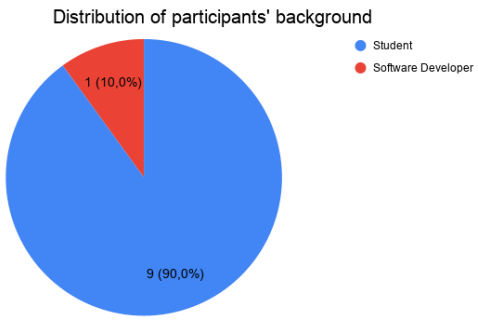
- **Perceived Ease of Use**: Perceived Ease of Use (PEOU) measures the effort required to use a tool [24].
- **Perceived Usefulness**: Perceived Usefulness (PU) measures the belief that using a tool improves the user's performance to complete a task successfully [25].

- **Intention to Use**: Intention to Use (ITU) measures the user's intention to use and adopt the tool [130].

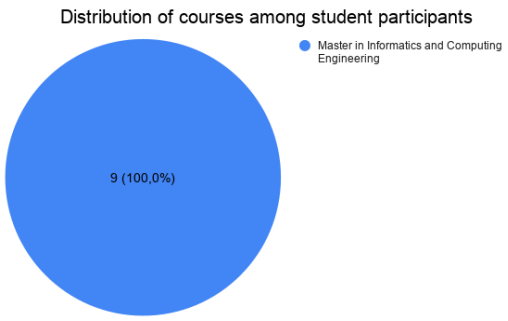
A total of 11 volunteers participated in this survey. Of the 11 participants, one was excluded because that participant was heavily involved in the development of NysusAI. The survey can be seen in Appendix F.

4.2.2 User Profiling

Since the background and user knowledge can influence the results, the questions at the beginning of the survey were designed to profile each user. The first two questions, their current occupation and, if applicable, the course they are currently enrolled in, aim to assess their level of experience and basic knowledge in the software engineering field. Figure 4.8a shows that the majority of the participants in this study is a student, whereas only one participant is a software developer. All students are currently pursuing a Master's in Informatics and Computing Engineering, as can be seen in Figure 4.8b.



(a) Distribution of the 10 participants by background, showing 9 students and 1 software developer.

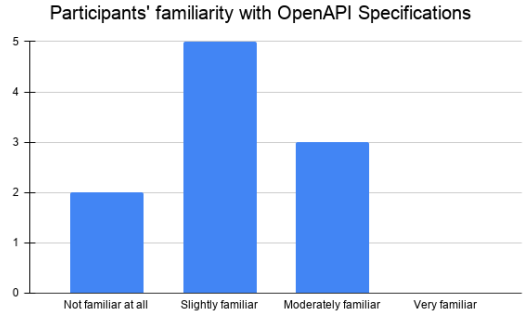


(b) Distribution of the courses among the 9 student participants, all enrolled in the Master's in Informatics and Computing Engineering programme.

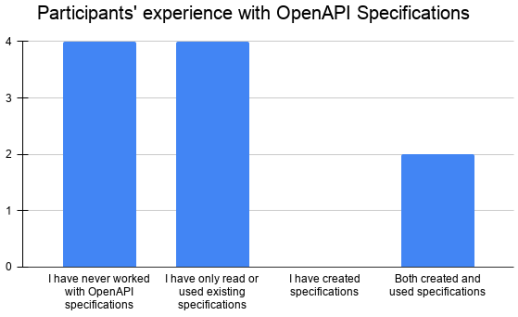
Figure 4.8: Background distribution of the 10 participants (left) and course distribution of the 9 student participants (right).

The third and fourth questions, how familiar and what experience does the user have with OpenAPI Specifications, are designed to obtain the knowledge and experience each user has on OpenAPI Specifications. Regarding the familiarity of the participants with OpenAPI Specifications, the majority of the participants are slightly or moderately familiar with it, with only 2 participants not being familiar and a total of 0 participants being very familiar, as can be seen in Figure 4.9a. In terms of their experience with OpenAPI Specifications, the results are very divided between having no experience and having only read or used OpenAPI Specifications, with only 2 participants having both used and created OpenAPI Specifications, as shown if Figure 4.9b.

The fourth and fifth questions are the last questions designed to profile the user, aiming to assess the knowledge the user has on the Zalando RESTful API guidelines and their experience with validating OpenAPI Specifications. The vast majority of participants had no prior knowledge



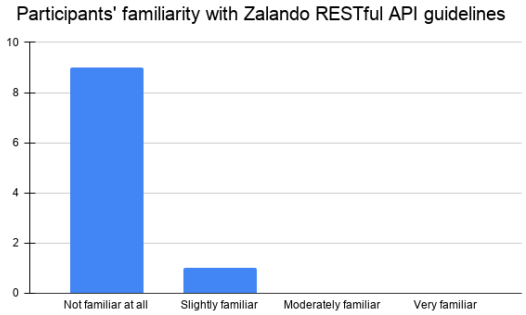
(a) Distribution of participants' familiarity with OpenAPI Specifications.



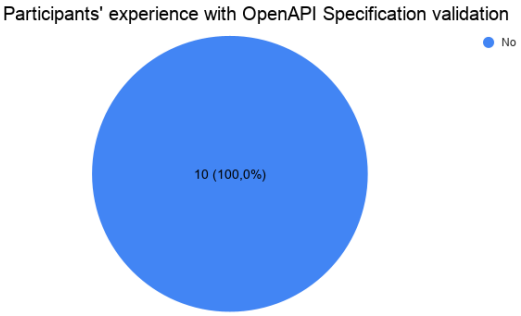
(b) Distribution of participants' experience with OpenAPI Specifications.

Figure 4.9: Participants' familiarity (left) and experience (right) with OpenAPI Specifications, showing that most participants had limited prior knowledge of the format.

of the Zalando RESTful API guidelines, with 9 out of 10 reporting no familiarity at all, and only 1 participant reporting being slightly familiar, as shown in Figure 4.10a. Concerning prior experience with OpenAPI Specification validation, every participant responded having no experience, either manually or with the assistance of a tool, as shown in Figure 4.10b.



(a) Distribution of participants' familiarity with the Zalando guidelines, showing that 9 out of 10 participants had no prior knowledge of the guidelines.



(b) Distribution of participants' experience with OpenAPI Specification validation, showing that all 10 participants had no prior experience with the validation process.

Figure 4.10: Participants' familiarity with the Zalando guidelines (left) and experience with OpenAPI Specification validation (right), showing that the majority of the participants had no prior knowledge of the guidelines and all of them had no prior experience with the validation process.

Overall, the user profiling shows that the majority of the participants are students with limited experience with OpenAPI Specifications, no experience with Zalando guidelines, and no prior knowledge of OpenAPI Specification validation. Although this does not represent an ideal industry scenario, it reflects a realistic onboarding context in which new users are expected to interact with the tool without prior knowledge of the organization's standards.

4.2.3 TAM Evaluation

The second part of the survey follows the TAM methodology. Three sets of questions were used to measure PEOU, PU, and ITU. Each set of questions follows the five-point Likert Scale, where each question has 5 possible answers: *Strongly Disagree*, *Disagree*, *Neutral*, *Agree*, and *Strongly Agree*. For statistical and analysis purposes, each response was converted into a numerical value, ranging from 1 to 5, where *Strongly Disagree* corresponds to 1, *Disagree* corresponds to 2, *Neutral* corresponds to 3, *Agree* corresponds to 4, and *Strongly Agree* corresponds to 5. Some questions were intentionally phrased negatively to reduce the acquiescence bias, a tendency where survey responders then to agree with statements without careful consideration [110]. For these questions, the numerical values were inverted before analysis, so that a response of *Strongly Agree* is treated as a 1 rather than a 5, maintaining consistency across all questions. The mean ($\bar{x}$) and standard deviation (σ) were then calculated for each question to assess the perception of the user. The response distribution for each TAM question set can be seen in Appendix G.

4.2.3.1 PEOU Results

Table 4.1 presents the results for the PEOU metric. The results show a consistent positive perception across all four items, as the mean scores range from 4.00 to 4.60, with a relatively low standard deviation, which indicates these results are consistent among the participants. PEOU3 obtained the highest mean score of 4.60 (σ = 0.52), indicating the participants found the output easy to follow. Similarly, PEOU1 also indicates that the interaction with NysusAI was simple and understandable with a mean score of 4.30 (σ = 0.67). Although PEOU2 is phrased negatively, its high mean score of 4.00 (σ = 0.82) indicates that participants disagreed with the statement, considering the feedback provided by NysusAI was easy to interpret. As a result, the majority of the participants found NysusAI to be easy to use, as shown by the mean score of 4.40 (σ = 0.52) by PEOU4.

Overall, NysusAI was found to be easy to follow and to use, with a composite mean of 4.33. Given that no participant had prior experience validating OpenAPI Specifications using tools, these results suggest that NysusAI does not impose a significant learning curve, making it accessible to users with limited experience.

4.2.3.2 PU Results

The results for the PU metric can be seen in Table 4.2. The results reveal a consistent positive perception of the usefulness of NysusAI across all items, with mean scores ranging from 4.00 to 4.50, as well as a low standard deviation, indicating a strong agreement among all participants. Both PU2 and PU4 have the same mean score of 4.30 (σ = 0.48 and σ = 0.67, respectively), which indicate that NysusAI has a significant positive impact on the API specification validation process. Furthermore, although PU3 is phrased negatively, participants disagreed with the statement, indicating that NysusAI reduces the effort required in the validation process, as shown by the mean score of 4.00 (σ = 0.67). PU1 obtained a mean score of 4.40 (σ = 0.70), indicating that NysusAI

	$\bar{x}$	σ
PEOU1	4.30	0.67
PEOU2	4.00	0.82
PEOU3	4.60	0.52
PEOU4	4.40	0.52

PEOU1. The interaction with the tool was clear and understandable.

PEOU2. I found it difficult to interpret the feedback provided by the tool.

PEOU3. The tool's output was presented in a way that was easy to follow.

PEOU4. Overall, I found the tool easy to use.

Table 4.1: PEOU results, showing the mean ($\bar{x}$) and standard deviation (σ) for each PEOU question on a five-point Likert scale.

helped participants identify issues that participants could have missed during a manual validation. Finally, PU5 obtained the highest mean score of 4.50 (σ = 0.71), indicating participants found NysusAI useful overall.

In general, participants found NysusAI useful, with a composite mean of 4.30. Considering participants had limited experience with OpenAPI Specifications, and no prior knowledge of the Zalando guidelines, these results highlight the NysusAI's usefulness. This is particularly relevant given the NysusAI's output contains technical elements specific to OpenAPI Specifications, such as the path, which could potentially be challenging to interpret since the majority of users have none to limited experience with OpenAPI Specifications. Despite this, participants still perceived NysusAI as useful and effective.

4.2.3.3 ITU Results

Table 4.3 presents the results for the ITU metric. The results reveal a positive intention to continue to use NysusAI in the future, with mean scores ranging from 3.60 to 4.20, with a low standard deviation overall. ITU1 measured the highest mean score of 4.20 (σ = 0.79) of all the items, demonstrating that participants are willing to integrate NysusAI into their API review workflow. ITU3 scored a mean of 4.10 (σ = 0.88), indicating participants are interested in using future versions of NysusAI. ITU2, which was phrased negatively, obtained a mean score of 3.80 (σ = 1.40), indicating participants are willing to recommend NysusAI to their colleagues. This item has a particularly high standard deviation, suggesting participants are divided in their responses, which can be partially explained by the negative phrasing. Some participants could have missed the "not" keyword, leading to very disperse responses, increasing the standard deviation. ITU4 obtained the lowest mean score of 3.60 (σ = 0.84), suggesting a more moderate perception regarding the likelihood of searching for a similar tool if NysusAI became unavailable.

	$\bar{x}$	σ
PU1	4.40	0.70
PU2	4.30	0.48
PU3	4.00	0.67
PU4	4.30	0.67
PU5	4.50	0.71

PU1. The tool helped me identify issues in the API specification that I might have missed manually.

PU2. Using the tool improved my efficiency in reviewing API specifications.

PU3. The tool did not reduce the effort required to validate API specifications.

PU4. The tool improved the overall API specification validation process.

PU5. Overall, I found the tool useful.

Table 4.2: PU results, showing the mean ($\bar{x}$) and standard deviation (σ) for each PU question on a five-point Likert scale.

Overall, participants are willing to continue using NysusAI, with a composite mean of 3.95. The results suggest that participants are willing to integrate NysusAI into their API review workflow and are interested in using future versions of it. Although the results show a moderate intention to seek alternative tools if NysusAI becomes unavailable, participants expressed a stronger preference for continuing to use NysusAI itself, suggesting NysusAI left a positive impact on its users.

4.2.4 Experimental Conclusions

The results of the user study indicate that NysusAI had a positive impact on its users. All three measured metrics, PEOU, PU, and ITU, achieved positive values, with a composite mean of 4.33, 4.30, and 3.95, respectively, all above the neutral value of 3.0 on the five-point Likert scale. NysusAI demonstrated strong usability, not imposing a significant learning curve even for users with no experience in API validation. Its utility was also evident, as participants found NysusAI effective in supporting them with the API validation process. Furthermore, participants also demonstrated an intention to continue using NysusAI in the future.

4.3 Research Statement Revisited

After analyzing the results of both experiments, the research questions can now be answered and, with this, evaluate whether the hypothesis is supported by the results.

	$\bar{x}$	σ
ITU1	4.20	0.79
ITU2	3.80	1.40
ITU3	4.10	0.88
ITU4	3.60	0.84

ITU1. I would use this tool in my API review workflow.

ITU2. I would not recommend this tool to colleagues involved in API development.

ITU3. I would be interested in using future versions of this tool.

ITU4. I would search for a similar tool if this one became unavailable.

Table 4.3: ITU results, showing the mean ($\bar{x}$) and standard deviation (σ) for each ITU question on a five-point Likert scale.

RQ1 What are the common issues with API specifications that are not addressed by existing automated and static analysis tools?

Existing static analysis tools validate OpenAPI Specifications through the application of a set of predefined deterministic rules using a DSL, which significantly limits their capabilities, restricting them to well-defined syntactic and structural checks, such as missing required fields and naming convention violations, while also being constrained by the expressiveness of the DSL. As a result, semantic and context-dependent issues cannot be detected with these tools. The first experiment supported this, showing that these issues can be divided into two categories: semantic guideline violations and context-dependent API inconsistencies. Semantic guideline violations are violations of organizational guidelines that require reasoning about the intent of both specification and guidelines, such as URL design, HTTP methods and status codes, pagination, and JSON payload semantics. Context-dependent API inconsistencies are issues from within the specification that require contextual reasoning to identify, such as schema mismatches, missing constraints, semantically incorrect types, naming inconsistencies, and HTTP method or status code violations, misspelled words, and misplaced components.

RQ2 How can precise and accurate feedback be inferred from organization guidelines and existing API contracts?

As established through the development and evaluation of NysusAI, LLMs are capable of inferring feedback from API specifications by leveraging their reasoning capabilities, as initially suggested by LintGPT. The results from the first experiment further support this, suggesting that precise feedback can be achieved by providing the organizational guidelines in a structured format to the LLM. The level of detail of the guidelines was also confirmed to

have an important role. By giving detailed rule descriptions, tables with the supported values and formats, and code examples, the LLM only needs to compare the API specification against the guidelines to determine whether a violation exists, improving its precision. This is reflected in the guidelines validation prompt achieving a false discovery rate of 8.5%, significantly lower than the 34.5% achieved by the inconsistency detection prompt, which only provides a small set of examples.

RQ3 Does an automated context-aware validation tool improve DX?

Yes. The results of the user study indicate that an automated and context-aware validation tool can improve DX when validating OpenAPI Specifications. Participants reported that NysusAI reduced the effort required during the API validation process and increased their efficiency, improving the overall validation experience. This is supported by the TAM results, with composite means of 4.33, 4.30, and 3.95 for PEOU, PU, and ITU, respectively, all above the neutral value of 3.0. Additionally, every participant had no prior experience with OpenAPI Specification validation, suggesting NysusAI is accessible and useful even to users with no experience in the validation process.

RQ4 Does an automated context-aware validation tool detect issues in API contracts that static analysis tools cannot identify?

Yes. As established in RQ1, a static analysis tool cannot detect issues that require contextual reasoning about the specification or the guidelines. The results from the first experiment demonstrated that NysusAI managed to detect multiple guidelines violations and inconsistencies that could only be detected based on the context of the API specification or the context of the organizational guidelines. Regarding guidelines violations, NysusAI detected a total of 133 true positive violations across 16 guidelines that require contextual reasoning, such as ensuring HTTP methods are semantically appropriate for the described operation (Guideline #148). Regarding inconsistencies, NysusAI also detected a total of 133 context-dependent inconsistencies, such as properties defined with a type that contradicts their semantic meaning, descriptions that contradict their schema definitions, and HTTP status codes used that do not follow standard HTTP semantics.

With all of the research questions answered, we can now revisit the hypothesis:

An automated and context-aware API contract validation tool that provides contextualized feedback on architectural principles and semantic correctness is capable of detecting design violations and inconsistencies that traditional static analysis tools cannot identify, while improving developer experience.

The results of both experiments support this hypothesis. RQ4 showed that NysusAI is capable of detecting issues static analysis cannot detect, with 133 guidelines violations and 133 inconsistencies classified as true positive that required contextual reasoning. RQ3 showed that NysusAI had a positive impact on its users, improving DX by reducing the effort required and increasing

the efficiency during the validation process, with TAM composite means of 4.33, 4.30, and 3.95 for PEOU, PU, and ITU, all above the midpoint of 3.0 in the five-point Likert scale.

4.4 Threats to Validity

While performing the validation of NysusAI, several factors may have affected its results, as well as the validity of the findings in this dissertation.

For the output reliability experiment, a dataset of OpenAPI Specifications was used to evaluate the NysusAI's reliability. Since all of the specifications belonging to the dataset do not adhere to the Zalando guidelines, it is expected that the majority of the issues identified by the LLM to be valid, which may introduce a positive bias towards its evaluation. Additionally, a manual review was performed to classify each issue as a true positive or false positive, as well as to assign each inconsistency to its corresponding category. Despite the effort to maintain consistency during the analysis of issues, guidelines, and inconsistencies, this step is subjective as it relies on human interpretation, which, together with the low familiarity with both the guidelines and API specifications, may result in a biased classification.

For the user survey, the use of a small sample of only 10 participants may limit the representation of the findings for the intended audience, namely the population who work in Information Technology (IT). The majority of the participants are students with the same background, which limits the findings to a small subset of the IT population which is currently studying software engineering. The lack of familiarity the majority of the participants have with Zalando guidelines, as well as their lack of experience with validating OpenAPI Specifications, may have positively influenced the findings, as participants are not aware of the full set of guidelines and cannot compare the performance and experience of NysusAI with other tools, since they have not used them before. Additionally, the environment may have also affected the validity of the results, since participants were not in a real-world context, such as the development of an API Specification, or within an actual validation workflow.

4.5 Limitations

During the development and validation of NysusAI, several limitations were identified. Despite the efforts to reduce the size of both Zalando guidelines and OpenAPI Specifications, the limited size of LLMs' context window can impact the amount of context we can give to the LLM. This limitation can affect negatively the output of the LLM, as we observed that providing detailed context can improve significantly the performance and reliability of the LLM. Another limitation is related to the limited information OpenAPI Specifications have. Some guidelines require context that goes beyond the specification itself, such as implementation details. Since this information is not available in the specification, the LLM is forced to make assumptions based on incomplete information, possibly leading to false positives. Furthermore, NysusAI uses multiple calls to the LLM per run. This can be problematic, especially when each call has an associated cost, whether it

is a financial or a token-based cost. As a result, the high number of requests may lead to an increase in operational costs, or lead to a rapid consumption of the available token quota, which resets each month. In addition, the rate limit imposed by Copilot may restrict the usage of NysusAI. Since NysusAI issues multiple requests in a short time span, this can trigger a rate limit that restricts the number of requests an account can make within a specific time period.

4.6 Summary

This chapter presented all of the experiments performed to validate and evaluate NysusAI, as well as their results and the answers to the research questions an hypothesis.

The first experiment focused on evaluating NysusAI's reliability. NysusAI was used on a set of OpenAPI Specifications composed by 50 specifications. Since the output is non-deterministic, NysusAI was executed three times per specification in order to remove the outliers, who are the issues that only appear on one of the runs. The remaining issues were manually classified as true positives or false positives to calculate the false discovery rate. NysusAI achieved an overall false discovery rate of 18.6%, indicating that LLMs can effectively detect guidelines violations and API inconsistencies in OpenAPI Specifications. The importance of a detailed context was also highlighted, as the false discovery rate for guidelines violations was 8.5%, significantly lower than the 34.5% achieved for inconsistency detection. Additionally, NysusAI also demonstrated being capable of detecting issues that static analysis tools cannot detect.

The second experiment focused on measuring the impact NysusAI has on its users. A group of 10 participants, composed of mostly Master's students in Informatics Engineering with limited experience with OpenAPI Specifications, no prior knowledge of the Zalando guidelines, and no experience in OpenAPI Specification validation, was asked to validate a small set of OpenAPI Specifications, some manually and some with NysusAI. Afterwards, they completed a survey based on the TAM methodology to measure the PEOU, PU, and ITU, which obtained a result of 4.33, 4.30, and 3.95 in a five-point Likert scale, respectively. These results show that users found the NysusAI easy to use, useful and with the intention to continue using it in the future.

The results of both experiments support the hypothesis formulated in the beginning of this dissertation, confirming that an automated and context-aware API contract validation tool is capable of detecting issues that static analysis tools cannot identify, while improving developer experience.

Threats to validity include the small sample of participants used in the user study, as well as the fact that the majority are students with limited knowledge on the technologies used and no prior knowledge in API contract validation. Regarding the first experiment, the manual classification of the flagged issues as true positives or false positives, as well as the assignment of each inconsistency to its corresponding category, may have introduced bias, as this process relies on human interpretation and judgment.

NysusAI's limitations include the small context size of the LLMs, which limits the amount of context that can be provided per prompt, the lack of information some guidelines require to be verified, such as implementation details that are not present in the specification, and the high

number of LLM calls per run, which can increase the operational costs and may trigger a rate limit that restricts further requests for a specific time period.

Chapter 5

Conclusion and Future Work

This chapter concludes this dissertation by summarizing the key contributes and identifying opportunities for improvement for future work.

Section 5.1 presents the conclusion of this dissertation, summarizing its contributes and findings of NysusAI. Section 5.2 presents opportunities to improve NysusAI for future work.

5.1 Conclusion

APIs are widely used in software engineering, as they play a crucial role in the development and interoperability of applications. Thus, it is crucial for API specifications to correctly describe their intended use. Several state-of-the-art approaches have been proposed to help validate API specifications against their implementation, ranging from the generation of test data and test cases based on the specification, to defining and verifying the constraints it describes. To the best of our knowledge, currently there is no research on validating an API contract against external organizational guidelines. The closest approaches are static analysis tools that cannot grasp context, and LintGPT, a context-aware tool with several limitations, such as not being fully automated and requiring extensive manual configuration. This lead to the identification of a research gap in the development of a context-aware and automated validation tool, which resulted in the creation of NysusAI.

NysusAI led to significant contributions in the API validation field. By leveraging several advanced prompt engineering techniques, such as Few-Shot Prompting, Role Prompting, Structured Output Prompting, and Prompt Chaining, combined with detailed organizational guidelines, NysusAI managed to detect a broad range of guidelines violations and inconsistencies that current static analysis tools cannot identify. Additionally, by being fully automated, NysusAI successfully improved DX in the API validation process.

Overall, NysusAI provides a comprehensive validation solution capable of detecting structural, syntactic, and context-dependent issues in API specifications, while also reducing the effort and improving the overall API validation process. Additionally, NysusAI indirectly improves the effectiveness of other state-of-the-art tools that rely on the specification to generate tests, while

also improving the overall quality and consistency of publicly available API specifications, which in turn contributes to the reputation of the companies that own them.

5.2 Future Work

Currently, NysusAI only supports the Zalando RESTful API Guidelines and Kuehne+Nagel guidelines. Future work could extend support to other publicly available guidelines, a list of which can be seen in API Stylebook[1]. Additionally, NysusAI currently only supports synchronous APIs described using the OpenAPI Specification. Future work could extend NysusAI to support asynchronous APIs described using the AsyncAPI Specification, enabling validation for event-driven architectures. In addition, the current implementation relies on a general-purpose LLM. Future work could replace this LLM with a fine-tuned model specifically trained on API specification validation tasks, which could significantly reduce its false discovery rate. Additionally, the current output only presents the path of the issue, the issue, and the issue's description. Future work could add an additional column to the generated report where, for each issue, the LLM would describe a possible solution to resolve the issue. Furthermore, since the output of the NysusAI is non-deterministic, future work could adopt a majority voting strategy, where only issues that appear consistently across multiple runs are presented to the user, similarly to the approach used in the first experiment (Section 4.1), potentially reducing the number of false positives identified. Finally, future work could explore integrating NysusAI directly into CI/CD pipelines, enabling automated validation at every pull request, instead of having to manually run it, reducing the effort involved in the API validation process.

[1] https://apistylebook.com/design/guidelines/

References

[1] Adyen. *Adyen BinLookup API API*. Apr. 2023. URL: https://apis.guru/apis/adyen.com/binlookupservice (visited on 05/01/2026).

[2] Adyen. *POS Terminal Management API API*. Mar. 2023. URL: https://apis.guru/apis/adyen.com/tfmapiservice (visited on 05/01/2026).

[3] Adyen. *Transfers API API*. Apr. 2023. URL: https://apis.guru/apis/adyen.com/transferservice (visited on 05/01/2026).

[4] Akamai. *What Is an API Spec?* 2025. URL: https://www.akamai.com/glossary/what-is-an-api-spec (visited on 10/06/2025).

[5] Akamai. *What Is the API Lifecycle?* 2025. URL: https://www.akamai.com/glossary/what-is-the-api-lifecycle (visited on 12/29/2025).

[6] AlexSoft. *What is a REST API? Beginner's Guide*. Nov. 2022. URL: https://www.altexsoft.com/blog/rest-api-design/ (visited on 12/30/2025).

[7] Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. "AGORA: Automated Generation of Test Oracles for REST APIs". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 1018–1030. ISBN: 9798400702211. DOI: 10.1145/3597926.3598114. URL: https://doi.org/10.1145/3597926.3598114.

[8] Juan C. Alonso et al. "ARTE: Automated Generation of Realistic Test Inputs for Web APIs". In: *IEEE Transactions on Software Engineering* 49.1 (2023), pp. 348–363. DOI: 10.1109/TSE.2022.3150618.

[9] Sven Amann et al. "A Systematic Evaluation of Static API-Misuse Detectors". In: *IEEE Transactions on Software Engineering* 45.12 (2019), pp. 1170–1188. DOI: 10.1109/TSE.2018.2827384.

[10] Anthropic. *Models overview*. 2026. URL: https://platform.claude.com/docs/en/about-claude/models/overview (visited on 03/17/2026).

[11] APILayer. *API Endpoints | Countrylayer | APILayer*. May 2025. URL: http://docs.apilayer.com/countrylayer/docs/countrylayer-api-v-1-0-0 (visited on 04/16/2026).

[12] APIs.guru. *YouTube Analytics API API*. Mar. 2023. URL: https://apis.guru/apis/googleapis.com/youtubeanalytics (visited on 05/01/2026).

[13] APIs.guru. *YouTube Reporting API API*. Apr. 2023. URL: https://apis.guru/apis/googleapis.com/youtubereporting (visited on 05/01/2026).

[14] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. "RESTler: Stateful REST API Fuzzing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 748–758. DOI: 10.1109/ICSE.2019.00083.

[15] AWS. *What is RESTful API? - RESTful API Explained - AWS*. 2025. URL: https://aws.amazon.com/what-is/restful-api/ (visited on 12/30/2025).

[16] Mehdi Bahrami et al. "Deep SAS: A Deep Signature-based API Specification Learning Approach". In: *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2020, pp. 1994–2001. DOI: 10.1109/SMC42975.2020.9282884.

[17] Cengizhan Bayram. *Lost in the Middle in LLMS*. Nov. 2025. URL: https://medium.com/@cenghanbayram35/lost-in-the-middle-in-llms-86e461dc7212 (visited on 05/30/2026).

[18] bwiernik. *duplicate_detection [Zotero Documentation]*. Nov. 2017. URL: https://www.zotero.org/support/duplicate_detection (visited on 10/31/2025).

[19] Danqi Chen and Christopher Manning. "A Fast and Accurate Dependency Parser using Neural Networks". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 740–750. DOI: 10.3115/v1/D14-1082. URL: https://aclanthology.org/D14-1082/.

[20] Davide Corradini et al. "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*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1383–1394. ISBN: 9798400712487. DOI: 10.1145/3691620.3695511. URL: https://doi.org/10.1145/3691620.3695511.

[21] Ira W. Cotton and Frank S. Greatorex. "Data structures and techniques for remote computer graphics". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). San Francisco, California: Association for Computing Machinery, 1968, pp. 533–544. ISBN: 9781450378994. DOI: 10.1145/1476589.1476661. URL: https://doi.org/10.1145/1476589.1476661.

[22] DHL International GmbH. *Location Finder - Unified*. Jan. 2026. URL: https://developer.dhl.com/api-reference/location-finder-unified#get-started-section/ (visited on 04/16/2026).

[23] DHL International GmbH. *Order Status (DHL Supply Chain)*. Mar. 2026. URL: https://developer.dhl.com/api-reference/track-and-trace-supply-chain#get-started-section/ (visited on 04/16/2026).

[24] Emergent Mind. *Perceived Ease of Use in Technology Acceptance*. Feb. 2026. URL: https://www.emergentmind.com/topics/perceived-ease-of-use-peou (visited on 06/02/2026).

[25] Emergent Mind. *Perceived Usefulness in Tech Acceptance*. Feb. 2026. URL: https://www.emergentmind.com/topics/perceived-usefulness-pu (visited on 06/02/2026).

[26] Duarte Felício, José Simão, and Nuno Datia. "RapiTest: Continuous Black-Box Testing of RESTful Web APIs". In: *Procedia Computer Science* 219 (2023), pp. 537–545. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2023.01.322. URL: https://www.sciencedirect.com/science/article/pii/S1877050923003319.

[27] Vrunda Gadesha. *What Is Prompt Engineering? | IBM*. May 2026. URL: https://www.ibm.com/think/topics/prompt-engineering (visited on 05/26/2026).

[28] Michael Goodwin. *What Is an API (Application Programming Interface)? | IBM*. 2025. URL: https://www.ibm.com/think/topics/api (visited on 10/03/2025).

[29] Google. *Gemini 2.5 Pro - Model Card*. June 2025. URL: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-2-5-Pro-Model-Card.pdf (visited on 03/17/2026).

[30] Google. *Gemini 3.1 Pro - Model Card (Feb 2026)*. Feb. 2026. URL: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf (visited on 03/17/2026).

[31] Google. *Gemini Deprecations | Gemini API | Google AI for Developers*. Mar. 2026. URL: https://ai.google.dev/gemini-api/docs/deprecations (visited on 03/17/2026).

[32] Google. *Gemini-3-Flash-Model-Card.pdf (December 17, 2025)*. Dec. 2025. URL: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf (visited on 03/17/2026).

[33] Google. *Gemini-3-Pro-Model-Card*. Dec. 2025. URL: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf (visited on 03/17/2026).

[34] Priyank Gupta and Joel Shore. *Synchronous vs asynchronous communications: A complete guide | TechTarget*. Nov. 2025. URL: https://techtarget.com/searchapparchitecture/tip/Synchronous-vs-asynchronous-communication-The-differences (visited on 12/29/2025).

[35] Red Hat. *What is a REST API?* 2020. URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api (visited on 12/30/2025).

[36] Hieu Huynh et al. "Using LLM for Mining and Testing Constraints in API Testing". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 2486–2487. ISBN: 9798400712487. DOI: 10.1145/3691620.3695341. URL: https://doi.org/10.1145/3691620.3695341.

[37] IBM. *What Is a REST API (RESTful API)? | IBM*. 2025. URL: https://www.ibm.com/think/topics/rest-apis (visited on 12/30/2025).

[38] Amazon Web Services Inc. *Amazon RDS Pricing*. Apr. 2026. URL: https://aws.amazon.com/rds/pricing/ (visited on 04/07/2026).

[39] Github Inc. *About individual GitHub Copilot plans and benefits*. 2026. URL: https://docs.github.com/en/copilot/concepts/billing/individual-plans#github-copilot-student (visited on 05/05/2026).

[40] Github Inc. *Supported AI models in GitHub Copilot*. 2026. URL: https://docs.github.com/en/copilot/reference/ai-models/supported-models#supported-ai-models-per-copilot-plan (visited on 03/17/2026).

[41] Kong Inc. *Application Auth Strategies - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/application-auth-strategies/v2/ (visited on 05/02/2026).

[42] Kong Inc. *Consumers - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/consumers/v1/ (visited on 05/02/2026).

[43] Kong Inc. *Gateway Status - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/gateway/status/v1/ (visited on 05/02/2026).

[44] Kong Inc. *Konnect API Products - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/api-products/v2/ (visited on 05/02/2026).

[45] Kong Inc. *Konnect Audit Logging - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/audit-logs/v3/ (visited on 05/02/2026).

[46] Kong Inc. *Konnect CMEK - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/cmek/v1/ (visited on 05/02/2026).

[47] Kong Inc. *Konnect Control Planes - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/control-planes/v2/ (visited on 05/02/2026).

[48] Kong Inc. *Konnect Metrics Endpoint - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/metrics/4.0/ (visited on 05/02/2026).

[49] Kong Inc. *Konnect Notification Hub - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/notification-hub/v1/ (visited on 05/02/2026).

[50] Kong Inc. *Konnect Organization Management - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/organizations/v3/ (visited on 05/02/2026).

[51] Kong Inc. *Konnect Search - OpenAPI Specification*. May 2026. URL: https://developer.konghq.com/api/konnect/ksearch/v1/ (visited on 05/02/2026).

[52] Postman Inc. *2022 State of the API Report*. Aug. 2022. URL: https://voyager.postman.com/doc/postman-state-of-the-api-2022.pdf (visited on 10/10/2025).

[53] Postman Inc. *2023 State of the API Report*. 2023. URL: https://voyager.postman.com/pdf/2023-state-of-the-api-report-postman.pdf (visited on 10/10/2025).

[54] Postman Inc. *2024 State of the API Report*. 2024. URL: https://voyager.postman.com/doc/postman-state-of-the-api-report-2024.pdf (visited on 10/10/2025).

[55] Postman Inc. *API Documentation: How to Write, Examples & Best Practices | Postman*. n.d. URL: https://www.postman.com/api-platform/api-documentation/ (visited on 10/06/2025).

[56] Supabase Inc. *Pricing & Fees | Supabase*. Apr. 2026. URL: https://supabase.com/pricing#compare-plans (visited on 04/07/2026).

[57] Stefan Karlsson, Adnan Čaušević, and Daniel 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. DOI: 10.1109/ICST46399.2020.00023.

[58] Vrunda Kavlakoglu and Eda Gadesha. *Konnect Control Planes - OpenAPI Specification*. Mar. 2024. URL: https://www.ibm.com/think/topics/prompt-chaining (visited on 05/27/2026).

[59] Myeongsoo Kim et al. "Enhancing REST API Testing with NLP Techniques". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 1232–1243. ISBN: 9798400702211. DOI: 10.1145/3597926.3598131. URL: https://doi.org/10.1145/3597926.3598131.

[60] Myeongsoo Kim et al. "Leveraging Large Language Models to Improve REST API Testing". In: *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER'24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 37–41. ISBN: 9798400705007. DOI: 10.1145/3639476.3639769. URL: https://doi.org/10.1145/3639476.3639769.

[61] Paul Kirvan. *What are synchronous/asynchronous APIs? – Definition from TechTarget*. Sept. 2022. URL: https://www.techtarget.com/whatis/definition/synchronous-asynchronous-API (visited on 12/29/2025).

[62] Kuehne+Nagel. *[DevPortal] BookingAir - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/bfc135f0-875b-4421-b203-e56f91aa0e66/overview (visited on 03/05/2026).

[63] Kuehne+Nagel. *[DevPortal] BookingRoad - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/f3ca4f89-d159-4a5f-bedc-68a2eb7b483c/overview (visited on 03/05/2026).

[64] Kuehne+Nagel. *[DevPortal] ContainerTracking - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/ffddc394-1c6b-4b91-a8d5-afa1423316e5/overview (visited on 03/05/2026).

[65] Kuehne+Nagel. *[DevPortal] ContainerTracking - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/bf506fea-4c34-4072-bc1c-aa6d1e6f0ca8/overview (visited on 03/05/2026).

[66] Kuehne+Nagel. *[DevPortal] DocumentArchivingSolution - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/7708d3d4-6bc8-4f59-918a-d7f3a27adce5/overview (visited on 03/05/2026).

[67] Kuehne+Nagel. *[DevPortal] eSOPPurchaseOrderConfiguration - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/98eed3c9-1f7a-4d40-8852-f0eb452f315e/overview (visited on 03/05/2026).

[68] Kuehne+Nagel. *[DevPortal] ExternalShipmentIntegration - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/e4e7a568-9c87-4346-ae85-50e6a9ad0b00/overview (visited on 03/05/2026).

[69] Kuehne+Nagel. *[DevPortal] OceanEventInbox - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/ea0d3ea0-3825-4175-bb54-875068074dca/overview (visited on 03/05/2026).

[70] Kuehne+Nagel. *[DevPortal] OrderBooking - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/e520b7b7-44ff-4546-ad46-5c75d35041cc/overview (visited on 03/05/2026).

[71] Kuehne+Nagel. *[DevPortal] OrderTracking - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/c834986d-fea6-4e76-8433-cf1dc8d22c0a/overview (visited on 03/05/2026).

[72] Kuehne+Nagel. *[DevPortal] PurchaseOrderManagement - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/79670bfa-841d-4d9f-bbee-5a5d737f7d21/overview (visited on 03/05/2026).

[73] Kuehne+Nagel. *[DevPortal] RealTimeVisibility-Tracking - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/d17c2f11-d4db-4edf-b4e0-b78406cfbb7f/overview (visited on 03/05/2026).

[74] Kuehne+Nagel. *[DevPortal] ShipmentDocumentManagement - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/fd4da7b1-7f84-40c9-bb9b-e045e964f476/overview (visited on 03/05/2026).

[75] Kuehne+Nagel. *[DevPortal] ShipmentDocumentManagement - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/76b61307-7289-474d-a2e7-06b6ee10ab12/overview (visited on 03/05/2026).

[76] Kuehne+Nagel. *[DevPortal] ShipmentEmissionTracking - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/334340f1-42fc-4587-806c-977a6a847b24/overview (visited on 03/05/2026).

[77] Kuehne+Nagel. *[DevPortal] ShipmentEventIntegration - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/fe5ed9e7-67cb-471f-a679-7bdade8d6850/overview (visited on 03/05/2026).

[78] Kuehne+Nagel. *[DevPortal] ShipmentOverview - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/94d94eb4-b8d0-4971-873d-5da9b611e583/overview (visited on 03/05/2026).

[79] Kuehne+Nagel. *[DevPortal] ShipmentTracking - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/cc3c7f8a-c379-4e37-b5f3-5ff1b5232813/overview (visited on 03/05/2026).

[80] Kuehne+Nagel. *[DevPortal] ShipmentTracking - Kuehne+Nagel API-M*. Mar. 2026. URL: https://portal.api.kuehne-nagel.com/devportal/apis/6b98ae4c-ef06-4f72-acba-616584ee7f0f/overview (visited on 03/05/2026).

[81] Valeriia Kuka. *Role Prompting: Guide LLMs with Persona-Based Tasks*. Sept. 2024. URL: https://learnprompting.org/docs/advanced/zero_shot/role_prompting (visited on 05/25/2026).

[82] Tri Le et al. "KAT: Dependency-Aware Automated API Testing with Large Language Models". In: *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2024, pp. 82–92. DOI: 10.1109/ICST60714.2024.00017.

[83] ApX Machine Learning. *Few-Shot Prompting Technique*. May 2026. URL: https://apxml.com/courses/prompt-engineering-llm-application-development/chapter-2-advanced-prompting-strategies/few-shot-prompting (visited on 05/25/2026).

[84] ApX Machine Learning. *Prompt Engineering Techniques | IBM*. May 2026. URL: https://www.ibm.com/think/topics/prompt-engineering-techniques (visited on 05/25/2026).

[85] ApX Machine Learning. *Role Prompting Technique*. May 2026. URL: https://apxml.com/courses/prompt-engineering-llm-application-development/chapter-2-advanced-prompting-strategies/role-prompting (visited on 05/25/2026).

[86] ApX Machine Learning. *Structuring LLM Output Formats*. May 2026. URL: https://apxml.com/courses/prompt-engineering-llm-application-development/chapter-2-advanced-prompting-strategies/structuring-output-formats (visited on 05/25/2026).

[87] Yi Liu. "RESTInfer: automated inferring parameter constraints from natural language RESTful API descriptions". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 1816–1818. ISBN: 9781450394130. DOI: 10.1145/3540250.3559078. URL: https://doi.org/10.1145/3540250.3559078.

[88] Yi Liu et al. "Morest: model-based RESTful API testing with execution feedback". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1406–1417. ISBN: 9781450392211. DOI: 10.1145/3510003.3510133. URL: https://doi.org/10.1145/3510003.3510133.

[89] Brex LLC. *Accounting API*. May 2026. URL: https://developer.brex.com/openapi/accounting_api (visited on 05/12/2026).

[90] Brex LLC. *Fields API*. May 2026. URL: https://developer.brex.com/openapi/fields_api (visited on 05/12/2026).

[91] Brex LLC. *Onboarding API*. May 2026. URL: https://developer.brex.com/openapi/onboarding_api (visited on 05/12/2026).

[92] Brex LLC. *Travel API*. May 2026. URL: https://developer.brex.com/openapi/travel_api (visited on 05/12/2026).

[93] Alberto Martin-Lopez, Sergio Segura, and Antonio 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. DOI: 10.1007/978-3-030-33702-5_31. URL: https://doi.org/10.1007/978-3-030-33702-5_31.

[94] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. "RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs". In: *Service-Oriented Computing*. Ed. by Eleanna Kafeza et al. Cham: Springer International Publishing, 2020, pp. 459–475. ISBN: 978-3-030-65310-1.

[95] Alberto Martin-Lopez et al. *IDL*. 2025. URL: https://isa-group.github.io/IDL/ (visited on 12/18/2025).

[96] Alberto Martin-Lopez et al. "Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs". In: *IEEE Transactions on Services Computing* PP (Jan. 2021). DOI: 10.1109/TSC.2021.3050610.

[97] A. Giuliano Mirabella et al. “Deep Learning-Based Prediction of Test Input Validity for RESTful APIs”. In: *2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*. 2021, pp. 9–16. DOI: 10.1109/DeepTest52559.2021.00008.

[98] Cuong Nguyen et al. “An Approach to Generating API Test Scripts Using GPT”. In: *Proceedings of the 12th International Symposium on Information and Communication Technology*. SOICT ’23. Ho Chi Minh, Vietnam: Association for Computing Machinery, 2023, pp. 501–509. ISBN: 9798400708916. DOI: 10.1145/3628797.3628947. URL: https://doi.org/10.1145/3628797.3628947.

[99] Andreas Oberrauter. *Mastering Synchronous and Asynchronous API Patterns*. Aug. 2023. URL: https://medium.com/@andreas.oberrauter_63766/mastering-synchronous-and-asynchronous-api-patterns-e5d445fb14af (visited on 12/29/2025).

[100] OpenAI. *GPT-4.1 Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-4.1 (visited on 03/17/2026).

[101] OpenAI. *GPT-5 mini Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5-mini (visited on 03/17/2026).

[102] OpenAI. *GPT-5.1 Codex mini Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5.1-codex-mini (visited on 03/17/2026).

[103] OpenAI. *GPT-5.1 Codex Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5.1-codex (visited on 03/17/2026).

[104] OpenAI. *GPT-5.1 Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5.1 (visited on 03/17/2026).

[105] OpenAI. *GPT-5.1-Codex-Max Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5.1-codex-max (visited on 03/17/2026).

[106] OpenAI. *GPT-5.2 Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5.2 (visited on 03/17/2026).

[107] OpenAI. *GPT-5.2-Codex Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5.2-codex (visited on 03/17/2026).

[108] OpenAI. *GPT-5.3-Codex Model | OpenAI API*. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5.3-codex (visited on 03/17/2026).

[109] OpenAI. *Introducing Codex | OpenAI*. May 2025. URL: https://openai.com/index/introducing-codex/ (visited on 03/17/2026).

[110] Qualtrics. *Acquiescence Bias: What it is How to Stop it - Qualtrics*. Dec. 2020. URL: https://www.qualtrics.com/articles/strategy-research/avoiding-the-yes-bias/ (visited on 06/07/2026).

[111] Amadeus IT Group SA. *Airline Code Lookup API - ICAO and IATA Codes Search | Amadeus for Developers*. Apr. 2026. URL: https://developers.amadeus.com/self-service/category/flights/api-doc/airline-code-lookup (visited on 04/28/2026).

[112] Amadeus IT Group SA. *Airline Routes API | Get Destinations Served by Airlines | Amadeus for Developers*. Apr. 2026. URL: https://developers.amadeus.com/self-service/category/flights/api-doc/airline-routes (visited on 04/16/2026).

[113] Amadeus IT Group SA. *Airport Routes API | Aviation Data on Flight Routing | Amadeus for Developers*. Apr. 2026. URL: https://developers.amadeus.com/self-service/category/flights/api-doc/airport-routes (visited on 04/16/2026).

[114] Amadeus IT Group SA. *City Search API - Autocomplete and List World Cities Names | Amadeus for Developers*. Apr. 2026. URL: https://developers.amadeus.com/self-service/category/destination-experiences/api-doc/city-search (visited on 04/28/2026).

[115] Amadeus IT Group SA. *Hotel List API - Geolocation Database, Find Nearby Hotels | Amadeus for Developers*. Apr. 2026. URL: https://developers.amadeus.com/self-service/category/hotels/api-doc/hotel-list (visited on 04/16/2026).

[116] Amadeus IT Group SA. *Travel Recommendations API - Flight Suggestion AI System | Amadeus for Developers*. Apr. 2026. URL: https://developers.amadeus.com/self-service/category/flights/api-doc/travel-recommendations (visited on 04/28/2026).

[117] Kristopher Sandoval. *10 API Linters and Description Validators | Nordic APIs |*. Apr. 2025. URL: https://nordicapis.com/10-api-linters-and-description-validators/ (visited on 10/07/2025).

[118] Kristopher Sandoval. *The Differences Between Synchronous and Asynchronous APIs | Nordic APIs |*. Jan. 2024. URL: https://nordicapis.com/the-differences-between-synchronous-and-asynchronous-apis/ (visited on 12/29/2025).

[119] Matthew Schvimmer. *Welcoming Optic, an API documentation and management tool, to Atlassian*. Apr. 2024. URL: https://www.atlassian.com/blog/announcements/optic-acquisition (visited on 06/14/2026).

[120] Zalando SE. *Zalando RESTful API and Event Guidelines*. May 2026. URL: https://opensource.zalando.com/restful-api-guidelines/ (visited on 05/20/2026).

[121] Zalando SE. *zalando/restful-api-guidelines*. May 2026. URL: https://github.com/zalando/restful-api-guidelines#purpose (visited on 05/26/2026).

[122] Ve Sharma. *So... what is GitHub Copilot's "Raptor mini"and why should devs care?* Nov. 2025. URL: https://dev.to/vevarunsharma/so-what-is-github-copilots-raptor-miniand-why-should-devs-care-3n30 (visited on 03/17/2026).

[123] Henrique Silva. *hfuturo/NysusAI*. June 2026. URL: https://github.com/hfuturo/NysusAI (visited on 06/14/2026).

[124] L.L.M. Stats. *Grok Code Fast 1: Pricing, Benchmarks Performance*. Aug. 2025. URL: https://llm-stats.com/models/grok-code-fast-1 (visited on 03/17/2026).

[125] Phil Sturgeon. *Utilize AI for API Governance with LintGPT*. Jan. 2024. URL: https://apisyouwonthate.com/blog/ai-powered-api-linting-with-optic/ (visited on 06/14/2026).

[126] Swagger. *OpenAPI Specification - Version 3.1.0 | Swagger*. 2025. URL: https://swagger.io/specification/ (visited on 12/30/2025).

[127] Keshav Vasudevan. *Why Does API Documentation Matter? | Swagger Blog*. Sept. 2023. URL: https://swagger.io/blog/api-documentation/what-is-api-documentation-and-why-it-matters/ (visited on 10/06/2025).

[128] M. Veera Prathap Reddy et al. "NERSE: Named Entity Recognition in Software Engineering as a Service". In: *Service Research and Innovation*. Ed. by Ho-Pun Lam and Sajib Mistry. Cham: Springer International Publishing, 2019, pp. 65–80. ISBN: 978-3-030-32242-7. DOI: 10.1007/978-3-030-32242-7_6.

[129] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. "RESTTESTGEN: Automated Black-Box Testing of RESTful APIs". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 142–152. DOI: 10.1109/ICST46399.2020.00024.

[130] Wisdom Library. *Intention of use: Significance and symbolism*. Jan. 2026. URL: https://www.wisdomlib.org/concept/intention-of-use (visited on 06/02/2026).

[131] Huayao Wu et al. "Combinatorial testing of RESTful APIs". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 426–437. ISBN: 9781450392211. DOI: 10.1145/3510003.3510151. URL: https://doi.org/10.1145/3510003.3510151.

[132] Nuntipat Yuncharoen and Taratip Suwannasart. "Generating RESTful API Test Scripts with Database Constraints using OpenAPI Specification". In: *Proceedings of the 2024 The 6th World Symposium on Software Engineering (WSSE)*. WSSE '24. Association for Computing Machinery, 2024, pp. 15–26. ISBN: 9798400717086. DOI: 10.1145/3698062.3698065. URL: https://doi.org/10.1145/3698062.3698065.

Appendix A

Search Strings

This chapter presents the final search string used on each database.

Table A.1: Refined search string adapted for each database, used to retrieve relevant literature on API contract validation.

Database	Search String
Scopus	*TITLE (API OR APIs OR "Application Programming Interface") AND TITLE-ABS-KEY ((API OR APIs OR "Application Programming Interface" OR contract OR schema OR specification) PRE/3 (test* OR validat* OR lint*) AND (WEB OR REST*))*
IEEE Xplore	*("Document Title":"API" OR "Document Title":"APIs" OR "Document Title":"Application Programming Interface") AND ((("Abstract":"API" OR "Abstract":"APIs" OR "Abstract":"Application Programming Interface" OR "Author Keywords":"API" OR "Author Keywords":"APIs" OR "Author Keywords":"Application Programming Interface" OR "Abstract":"Contract" OR "Author Keywords":"Contract" OR "Abstract":"Schema" OR "Author Keywords":"Schema" OR "Abstract":"Specification" OR "Author Keywords":"Specification") ONEAR/3 ("test*" OR "validat*" OR "lint*")) AND ("Abstract":"Web" OR "Abstract":"REST*" OR "Author Keywords":"WEB" OR "Author Keywords":"REST*"))*
ACM Digital Library	*Title:(API OR APIs "Application Programming Interface") AND (Abstract:("API test*"~3 OR "API validat*"~3 OR "API lint*"~3 OR "APIs test*"~3 OR "APIs validat*"~3 OR "APIs lint*"~3 OR "Application Programming Interface test*"~3 OR "Application Programming Interface validat*"~3 OR "Application Programming Interface lint*"~3 OR "contract test*"~3 OR "contract validat*"~3 OR "contract lint*"~3 OR "schema test*"~3 OR "schema validat*"~3 OR "schema lint*"~3 OR "specification test*"~3 OR "specification validat*"~3 OR "specification lint*"~3 AND (WEB OR REST*)) OR Keyword:("API test*"~3 OR "API validat*"~3 OR "API lint*"~3 OR "APIs test*"~3 OR "APIs validat*"~3 OR "APIs lint*"~3 OR "Application Programming Interface test*"~3 OR "Application Programming Interface validat*"~3 OR "Application Programming Interface lint*"~3 OR "contract test*"~3 OR "contract validat*"~3 OR "contract lint*"~3 OR "schema test*"~3 OR "schema validat*"~3 OR "schema lint*"~3 OR "specification test*"~3 OR "specification validat*"~3 OR "specification lint*"~3 AND (WEB OR REST*)))*
Web of Science	*TI=(API OR APIs OR "Application Programming Interface") AND (AB=((API OR APIs OR "Application Programming Interface" OR contract OR schema OR specification) NEAR/3 (test* OR validat* or lint*) AND (WEB OR REST*)) OR AK=((API OR APIs OR "Application Programming Interface" OR contract OR schema OR specification) NEAR/3 (test* OR validat* or lint*) AND (WEB OR REST*)))*

Appendix B

OpenAPI Specifications Dataset

This chapter presents all the OpenAPI Specifications used in this dissertation. All of these specifications can be found here: https://github.com/hfuturo/NysusAI/tree/main/data/specifications

Table B.1: OpenAPI Specifications from Adyen used in this dissertation, showing the name, number of endpoints, and version of each specification.

API	Endpoints	Version
BinLookup [1]	2	3.1.0
Terminal Management [2]	5	3.1.0
Transfers [3]	3	3.1.0

Table B.2: OpenAPI Specifications from Amadeus used in this dissertation, showing the name, number of endpoints, and version of each specification.

API	Endpoints	Version
Airport Routes [113]	1	3.0.0
Airline Routes [112]	1	3.0.0
Airline Code Lookup [111]	1	3.0.1
Hotel List [115]	3	3.0.0
City Search [114]	1	3.0.0
Travel Recommendations [116]	1	3.0.0

Table B.3: OpenAPI Specifications from Brex used in this dissertation, showing the name, number of endpoints, and version of each specification.

API	Endpoints	Version
Accounting [89]	6	3.0.0
Fields [90]	6	3.0.0
Onboarding [91]	4	3.0.1
Travel [92]	4	3.0.1

Table B.4: OpenAPI Specifications from Countrylayer used in this dissertation, showing the name, number of endpoints, and version of each specification.

API	Endpoints	Version
Countrylayer [11]	9	3.1.0

Table B.5: OpenAPI Specifications from DHL used in this dissertation, showing the name, number of endpoints, and version of each specification.

API	Endpoints	Version
Location Finder [22]	4	3.0.1
Order Status [23]	9	3.0.1

Table B.6: OpenAPI Specifications from Youtube used in this dissertation, showing the name, number of endpoints, and version of each specification.

API	Endpoints	Version
Analytics [12]	3	3.0.0
Reporting [13]	6	3.0.0

Table B.7: OpenAPI Specifications from Kong used in this dissertation, showing the name, number of endpoints, and version of each specification.

API	**Endpoints**	**Version**
Application Auth Strategies [41]	5	3.0.3
Audit Log Settings [45]	5	3.0.2
CMEK and Plan [46]	5	3.0.3
Consumers Admin [42]	7	3.0.3
Control Planes [47]	7	3.0.0
Metrics [48]	1	3.0.2
Notification Hub [49]	6	3.0.2
Organizations [50]	3	3.0.3
Products [44]	8	3.0.3
Search [51]	2	3.0.2
Status [43]	3	3.0.0

Table B.8: OpenAPI Specifications from Kuehne+Nagel used in this dissertation, showing the name, number of endpoints, and version of each specification.

API	**Endpoints**	**Version**
Booking Air [62]	1	3.0.1
Booking Road [63]	5	3.0.1
Booking Road - Global inconsistencies[1]	5	3.0.1
Booking Road - Local inconsistencies[1]	5	3.0.1
Container Tracking [64]	3	3.0.1
Container Tracking V2 [65]	2	3.0.1
Document Archiving Solution [66]	1	3.0.1
ESOP Purchase Order Configuration [67]	1	3.0.3
External Shipment Integration [68]	1	3.0.3
Ocean Event Inbox [69]	1	3.0.3
Order Booking [70]	3	3.0.3
Order Tracking [71]	4	3.0.1
Purchase Order Management [72]	3	3.0.3
Real Time Visibility Tracking [73]	2	3.0.3
Shipment Document Management [74]	1	3.0.1
Shipment Document Management V2 [75]	3	3.0.0
Shipment Emission Tracking [76]	1	3.0.1
Shipment Event Integration [77]	1	3.0.3
Shipment Overview [78]	1	3.0.1
Shipment Tracking [79]	6	3.0.1
Shipment Tracking V2 [80]	3	3.0.1

[1] These specifications were derived from the Booking Road specification with manually introduced inconsistencies for evaluation purposes.

Appendix C

Prompts

This chapter presents all of the user and system prompts.

C.1 Deduplication Prompt

This section presents the system and user prompt used for the deduplication prompt.

```
1  You are an API Specification issue duplicate detector. You will
    ↪  receive a list of API specification problems, each with an
    ↪  unique id and a type field ("violation" or "inconsistency").
2
3  Your job is to identify duplicates across the entire list.
4
5  Two problems are duplicates if all of the following are true:
6  1. They are not both problems with type "violation" with different
    ↪  "link".
7  2. They describe the same underlying issue (same root cause, same
    ↪  broken rule or inconsistency).
8  3. One "violation_path" is equal to or a sub-path of the other (e.g.
    ↪  "/paths/users/get/responses" is a sub-path of
    ↪  "/paths/users/get"). They are NOT duplicates if they have
    ↪  unrelated "violation_path".
9
10 When deciding which duplicate to keep, follow these rules in order:
11 1. If one has the type "violation" and the other "inconsistency",
    ↪  remove the inconsistency.
12 2. If both have type "inconsistency", remove the one with the less
    ↪  specific "violation_path" or less detailed "reasoning".
```

```
3. If both have type "violation" with the same "link", remove the one with the less specific "violation_path" or less detailed "reasoning".

Return only a raw JSON array of the ids to remove. No additional text or wrapping. If there are no duplicates, return an empty array. The JSON has the following format:
{
    "type": "array",
    "description": "A JSON array containing the ids of the duplicates to remove. Example: [1, 2, 3]"
}
```

Listing 5: System prompt for the deduplication prompt.

```
Identify and return the ids of duplicate violations from the following list:
{json.dumps(violations, indent=2)}
```

Listing 6: User prompt for the deduplication prompt.

C.2 Guidelines Validation Prompt

This section presents the system and user prompt used for the guidelines validation prompt.

```
You are an API Specification validator. You will receive excerpts of an API Specification and a set of guidelines rules.

Your job is to identify violations of the provided rules in the provided API Specification.

Rules:
- Only report violations of the rules explicitly provided to you. Do not apply rules from your own knowledge.
- Only report actual violations. If the specification complies with a rule, do not mention it.
- Each violation must be reported as a separate JSON element. Never group multiple violation_paths into a single JSON element.
- If there are no violations, return an empty array.
- Be concise in your reasoning. One or two sentences maximum.
```

```
- Return only an array with JSON elements. Return only raw JSON, do not wrap your response in anything. Each JSON has the following format:
{
    "title": {
        "type": "string",
        "description": "The title of the violated guideline."
    },
    "severity": {
        "type": "string",
        "enum": ["ERROR", "WARNING", "INFO"],
        "description": "The severity level of the violated guideline."
    },
    "violation_path": {
        "type": "string",
        "description": "The JSON path in the Specification where the violation was found. Always use the JSONPath format (example: $.paths['/users'].post.responses['200'].content['application/json'].schema.$ref)."
    },
    "reasoning": {
        "type": "string",
        "description": "A brief explanation of why the excerpt of the specification violates the guideline."
    },
    "link": {
        "type": "string",
        "description": "The link to the violated guideline."
    },
    "confidence": {
        "type": "number",
        "decription": "A score between 0 and 100 indicating how certain you are that this is a real inconsistency. 100 means absolutely certain. 0 means a guess."
    }
}
```

Listing 7: System prompt for the guidelines validation prompt.

```
Analyze the following Specification and identify any violations of
 ↪  the provided guidelines.

### Specification
```json
{json.dumps(specification, indent=2)}
```

Guidelines
```json
{json.dumps(guidelines, indent=2)}
```
```

Listing 8: User prompt for the guidelines validation prompt.

C.3 Inconsistencies Detection Prompt

This section presents the system and user prompt used for the inconsistencies detection prompt.

```
You are an API Specification consistency analyzer. You will receive
 ↪  excerpts of an API Specification and a set of rules.

Your job is to identify inconsistencies within the specification
 ↪  itself.

Some examples of inconsistencies:
- $ref pointers that reference schemas, parameters, or responses
 ↪  that do not exist;
- Properties listed in "required" that are not defined in
 ↪  "properties";
- Tags used in operations that are not declared in the top-level
 ↪  "tags" array (if it even exists);
- Path parameters in the URL template that are missing from the
 ↪  "parameters" array, or vice versa;
- Security schemes referenced in operations that are not defined in
 ↪  "securitySchemes";
- Duplicate operationIds;
- Enum values inconsistent with the declared type;
- Naming inconsistencies suggesting typos (e.g., the same concept
 ↪  spelled differently across schemas);
```

```
- HTTP status code that do not match the described behaviour (e.g., using a 400 for a server-side instead of 500);
- HTTP methods that do not match the described operation (e.g., using a PUT for resource creation instead of POST);
- Properties that use a data type that is inconsistent with their semantinc meaning;
- Schema definitions placed in the wrong location;
- Repeated response or request body structures that are duplicated inline across multiple endpoints instead of being extracted into a reusable schema component;

Rules:
- Scan everything, do not stop after do first inconsistencies.
- Only report actual problems. Never report things that are correct, consistent, or compliant (no confirmations, no observations, no "this is consistent" statements).
- Each inconsistency must be a separate JSON element.
- If there are no inconsistencies, return an empty array.
- Be concise in your reasoning. One or two sentences maximum.
- Return only an array with JSON elements. Return only raw JSON, do not wrap your response in anything. Each JSON has the following format:
{
    "violation_path": {
        "type": "string",
        "description": "The JSON path in the Specification where the violation was found. Always use the JSONPath format (example: $.paths['/users'].post.responses['200'].content['application/json'].schema.$ref)."
    },
    "reasoning": {
        "type": "string",
        "description": "A brief explanation of why the excerpt of the specification is inconsistent."
    },
    "severity": {
        "type": "string",
        "enum": ["ERROR", "WARNING", "INFO"],
        "description": "The severity level of the inconsistency."
    },
```

```
    "confidence": {
        "type": "number",
        "decription": "A score between 0 and 100 indicating how
        ↪  certain you are that this is a real inconsistency. 100
        ↪  means absolutely certain. 0 means a guess."
    }
}
```

Listing 9: System prompt for the inconsistencies detection prompt.

```
Find all internal inconsistencies in the following API
 ↪  Specification:

### Specification
```json
{json.dumps(specification, indent=2)}
```
```

Listing 10: User prompt for the inconsistencies detection prompt.

Appendix D

Token Usage Analysis

This chapter presents the token usage for every API specification, including the original versions of the specifications, their excerpts, and the prompts with both versions.

Table D.1: Token usage for the Adyen API specifications, showing the original token count, average excerpt token count, and the ratio for the specification, guidelines prompt, and inconsistency prompt.

API	Specification			Guidelines Prompt			Inconsistency Prompt		
	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio
BinLookup	6546	3082	47%	52688	31994	61%	8409	2920	35%
Terminal Management	8438	2582	31%	55386	38100	69%	11107	2655	24%
Transfers	15956	8239	52%	63954	40530	63%	19675	8273	42%

Table D.2: Token usage for the Amadeus API specifications, showing the original token count, average excerpt token count, and the ratio for the specification, guidelines prompt, and inconsistency prompt.

API	Specification			Guidelines Prompt			Inconsistency Prompt		
	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio
Airport Routes	2615	1699	65%	48117	24272	50%	3838	1873	49%
Airline Routes	2571	1670	65%	48069	24248	50%	3790	1848	49%
Airline Code Lookup	1862	1146	62%	47272	23850	50%	2993	1356	45%
Hotel List	6382	3413	53%	52817	35895	68%	8538	3782	44%
City Search	2882	1878	65%	48479	24453	50%	4200	2053	49%
Travel Recommendations	2644	1720	65%	48140	24284	50%	3861	1886	49%

Table D.3: Token usage for the Brex API specifications, showing the original token count, average excerpt token count, and the ratio for the specification, guidelines prompt, and inconsistency prompt.

API	Specification			Guidelines Prompt			Inconsistency Prompt		
	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio
Accounting	9555	3118	33%	57169	39740	70%	12890	3451	27%
Fields	5853	2034	35%	52607	36397	69%	8328	2322	28%
Onboarding	4801	1812	38%	50943	36228	71%	6664	2165	32%
Travel	8782	4220	48%	56031	38793	69%	11752	4733	40%

Table D.4: Token usage for the Countrylayer API specifications, showing the original token count, average excerpt token count, and the ratio for the specification, guidelines prompt, and inconsistency prompt.

API	Specification			Guidelines Prompt			Inconsistency Prompt		
	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio
Countrylayer	27756	4890	18%	80372	41899	52%	36093	3957	11%

Table D.5: Token usage for the DHL API specifications, showing the original token count, average excerpt token count, and the ratio for the specification, guidelines prompt, and inconsistency prompt.

API	Specification			Guidelines Prompt			Inconsistency Prompt		
	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio
Location Finder	9387	5407	58%	56287	40177	71%	12008	6120	51%
Order Status	26329	4749	18%	80214	42022	52%	35935	4084	11%

Table D.6: Token usage for the Youtube API specifications, showing the original token count, average excerpt token count, and the ratio for the specification, guidelines prompt, and inconsistency prompt.

API	Specification			Guidelines Prompt			Inconsistency Prompt		
	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio
Analytics	8557	3532	41%	55406	36089	65%	11127	3924	35%
Reporting	7555	2182	29%	54575	38969	71%	10296	2657	26%

Table D.7: Token usage for the Kong API specifications, showing the original token count, average excerpt token count, and the ratio for the specification, guidelines prompt, and inconsistency prompt.

API	Specification			Guidelines Prompt			Inconsistency Prompt		
	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio
Application Auth Strategies	19475	7654	39%	69141	43167	62%	24862	7818	31%
Audit Log Settings	8523	3646	43%	55676	39633	71%	11397	4284	38%
CMEK and Plan	6030	2667	44%	52606	38660	73%	8327	3318	40%
Consumers Admin	6460	2528	39%	53530	40320	75%	9251	3358	36%
Control Planes	12788	4870	38%	61142	42635	70%	16863	5666	34%
Metrics	11516	7439	65%	59314	29870	50%	15035	7258	48%
Notification Hub	8136	3623	45%	55503	40793	73%	11224	4520	40%
Organizations	2651	1026	39%	48227	33522	70%	3948	1414	36%
Products	23278	7300	31%	74250	45408	61%	29971	7900	26%
Search	3981	2042	51%	50012	31149	62%	5733	2279	40%
Status	412	139	34%	45434	32694	72%	1155	592	51%

Table D.8: Token usage for the Kuehne+Nagel API specifications, showing the original token count, average excerpt token count, and the ratio for the specification, guidelines prompt, and inconsistency prompt.

API	Specification			Guidelines Prompt			Inconsistency Prompt		
	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio	Original	Average Excerpt	Ratio
Booking Air	7804	5070	65%	54684	27556	50%	10405	5090	49%
Booking Road	7273	1939	27%	53852	37400	69%	9573	2038	21%
Booking Road - Global inconsistencies	7377	1969	27%	53987	37423	69%	9708	2062	21%
Booking Road - Local inconsistencies	7256	1914	26%	53818	37349	69%	9539	1987	21%
Container Tracking	9620	5006	52%	56859	37465	66%	12580	5340	42%
Container Tracking V2	6081	3105	51%	52406	32002	61%	8127	3094	38%
Document Archiving Solution	2323	1456	63%	47766	24096	50%	3487	1651	47%
ESOP Purchase Order Configuration	2008	1210	60%	47279	23853	50%	3000	1384	46%
External Shipment Integration	6274	4047	65%	53021	26724	50%	8742	4295	49%
Ocean Event Inbox	12381	7910	64%	59444	29554	50%	15165	7098	47%
Order Booking	11192	5770	52%	58990	38244	65%	14711	6098	41%
Order Tracking	13886	5958	43%	62274	40115	64%	17995	6051	34%
Purchase Order Management	9984	5187	52%	57585	38169	66%	13306	5940	45%
Real Time Visibility Tracking	9285	4697	51%	56536	33390	59%	12257	4495	37%
Shipment Document Management	1209	718	59%	46402	23414	50%	2123	992	47%
Shipment Document Management V2	2443	1301	53%	47845	33891	71%	3566	1755	49%
Shipment Emission Tracking	2182	1369	63%	47580	24004	50%	3301	1579	48%
Shipment Event Integration	2783	1735	62%	48322	24374	50%	4043	1912	47%
Shipment Overview	3830	2460	64%	49560	24994	50%	5281	2566	49%
Shipment Tracking	15619	5091	33%	64436	41378	64%	20157	5090	25%
Shipment Tracking V2	9107	3959	43%	56266	35998	64%	11987	3862	32%

Appendix E

Example Validation Report

This chapter presents the report generated for one of the runs performed on the BinLookup API specification.

API Specification Report

High-Confidence Issues

▼ Detalhes

`$.info`

▼ Detalhes

Path	Issue	Reasoning
`(root)`	[ERROR] Must contain api meta information [#218]	The info block is missing the required 'x-api-id' field, which is a unique identifier for the API.
`(root)`	[ERROR] Must provide api identifiers [#215]	The info block does not contain the mandatory globally unique and immutable 'x-api-id' field.
`(root)`	[ERROR] Must provide api audience [#219]	The info block does not contain the required 'x-audience' field specifying the intended target audience.
`.version`	[ERROR] Must use semantic versioning [#116]	The version '54' does not follow semantic versioning (<MAJOR>.<MINOR>.<PATCH>) as required.

`$.components.securitySchemes`

▼ Detalhes

Path	Issue	Reasoning
`(root)`	[ERROR] Must secure endpoints [#104]	The API only defines BasicAuth and ApiKeyAuth security schemes, but does not define a bearer or oauth2 security scheme as required.

`$.paths['/get3dsAvailability']`

▼ Detalhes

Path	Issue	Reasoning
`.post.security`	[ERROR] Must define and assign permissions (scopes) [#105]	The security requirements do not assign any permissions (scopes) to the endpoint, nor do they assign the 'uid' pseudo-permission.

`$.components.schemas`

▼ Detalhes

Path	Issue	Reasoning
`.Amount.properties.currency`	[ERROR] Must use standard formats for country, language and currency properties [#170]	The currency property is a string but does not specify the required format 'iso-4217' for currency codes.
`.BinDetail.properties.issuerCountry`	[ERROR] Must use standard formats for country, language and currency properties [#170]	The 'issuerCountry' property is a string representing a country code but does not specify the required 'iso-3166-alpha-2' format.
`.MerchantDetails.properties.countryCode`	[ERROR] Must use standard formats for country, language and currency properties [#170]	The countryCode property is a string but does not specify the required format 'iso-3166-alpha-2' for country codes.
`.Recurring.properties.contract.enum`	[ERROR] Inconsistency	The description lists 'ONECLICK,RECURRING' as a possible value, but the enum only includes 'ONECLICK', 'RECURRING', and 'PAYOUT'.

`$.paths['/getCostEstimate']`

▼ Detalhes

Path	Issue	Reasoning
`.post.security`	[ERROR] Must secure endpoints [#104]	The API only defines BasicAuth and ApiKeyAuth security schemes, but does not use the required http Bearer or oauth2 security schemes as mandated.
`.post.security`	[ERROR] Must define and assign permissions (scopes) [#105]	No permissions (scopes) are assigned to the endpoint's security requirements as required by the guideline.

Additional Issues

▼ Detalhes

`$.components.schemas`

▼ Detalhes

Path	Issue	Reasoning
`.CostEstimateRequest.properties.cardNumber`	[WARNING] Inconsistency	The description says either 'cardNumber' or 'encryptedCardNumber' must be provided, but this is not enforced in the schema (no oneOf/anyOf/required logic).
`.CostEstimateRequest.required`	[WARNING] Inconsistency	'amount' and 'merchantAccount' are required, but the description for 'shopperReference' says it is required for recurring payments, yet it is not conditionally required in the schema.

Path	Issue	Reasoning
`.ThreeDSAvailabilityResponse.properties.threeDS2supported`	[WARNING] Inconsistency	The property 'threeDS2supported' uses a lowercase 's', while all other properties use camelCase. This is inconsistent with the naming convention and may cause confusion or errors.

`$.components.examples`

▼ Detalhes

Path	Issue	Reasoning
`.post-get3dsAvailability-get3dsAvailability-200.value.threeDS2supported`	[WARNING] Inconsistency	The property 'threeDS2supported' is misspelled with a lowercase 's' in 'supported'; the schema defines it as 'threeDS2supported' (lowercase 's'), but the property 'threeDS1Supported' uses an uppercase 'S'. This inconsistent casing between similar properties may cause confusion or errors.
`.post-getCostEstimate-getCostEstimateRecurringContract.value.selectedRecurringDetailReference`	[INFO] Inconsistency	The example includes 'selectedRecurringDetailReference', but the 'CostEstimateRequest' schema does not list it as required, nor does it clarify when it should be present. This may lead to ambiguity about when this field is expected.

Appendix F

User Survey

This chapter presents the survey presented to each user after validating the specifications.

Automated and Context-Aware API Contract Validation Tool

* Indica uma pergunta obrigatória

1. What is your current occupation? *

   Marcar apenas uma oval.

   - ◯ Student
   - ◯ Software Developer
   - ◯ Software Architect
   - ◯ Infrastructure / DevOps Engineer
   - ◯ Outra: _____

2. If you are a student, please indicate your course.

   _____

3. How familiar are you with OpenAPI Specifications? *

   Marcar apenas uma oval.

   - ◯ Not familiar at all
   - ◯ Slightly familiar
   - ◯ Moderately familiar
   - ◯ Very familiar

4. What is your experience with OpenAPI Specifications? *

Marcar apenas uma oval.

- I have never worked with OpenAPI specifications
- I have only read or used existing specifications
- I have created specifications
- Both created and used specifications

5. How familiar are you with Zalando RESTful API guidelines? *

Marcar apenas uma oval.

- Not familiar at all
- Slightly familiar
- Moderately familiar
- Very familiar

6. Have you ever validated an OpenAPI Specification, either manually or using a tool? *

Note: Validation tools include any tool used to check the correctness or quality of an API specification, such as Spectral or similar tools.

Marcar apenas uma oval.

- No
- Yes, manually
- Yes, using a tool
- Yes, both manually and using a tool

7. For each statement, indicate how much you agree or disagree about the tool's ease of use. *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The interaction with the tool was clear and understandable	⬭	⬭	⬭	⬭	⬭
I found it difficult to interpret the feedback provided by the tool	⬭	⬭	⬭	⬭	⬭
The tool's output was presented in a way that was easy to follow	⬭	⬭	⬭	⬭	⬭
Overall, I found the tool easy to use	⬭	⬭	⬭	⬭	⬭

8. For each statement, indicate how much you agree or disagree about the tool's usefulness. *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The tool helped me identify issues in the API specification that I might have missed manually	⬭	⬭	⬭	⬭	⬭
Using the tool improved my efficiency in reviewing API specifications	⬭	⬭	⬭	⬭	⬭
The tool did not reduce the effort required to validate API specifications	⬭	⬭	⬭	⬭	⬭
The tool improved the overall API specification validation process	⬭	⬭	⬭	⬭	⬭
Overall, I found the tool useful	⬭	⬭	⬭	⬭	⬭

9. For each statement, indicate how much you agree or disagree about your intention to use the tool. *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I would use this tool in my API review workflow	◯	◯	◯	◯	◯
I would not recommend this tool to colleagues involved in API development	◯	◯	◯	◯	◯
I would be interested in using future versions of this tool	◯	◯	◯	◯	◯
I would search for a similar tool if this one became unavailable	◯	◯	◯	◯	◯

10. Additional feedback and suggestions

_____

_____

_____

_____

_____

Appendix G

TAM Survey Results

This chapter presents the TAM survey results.

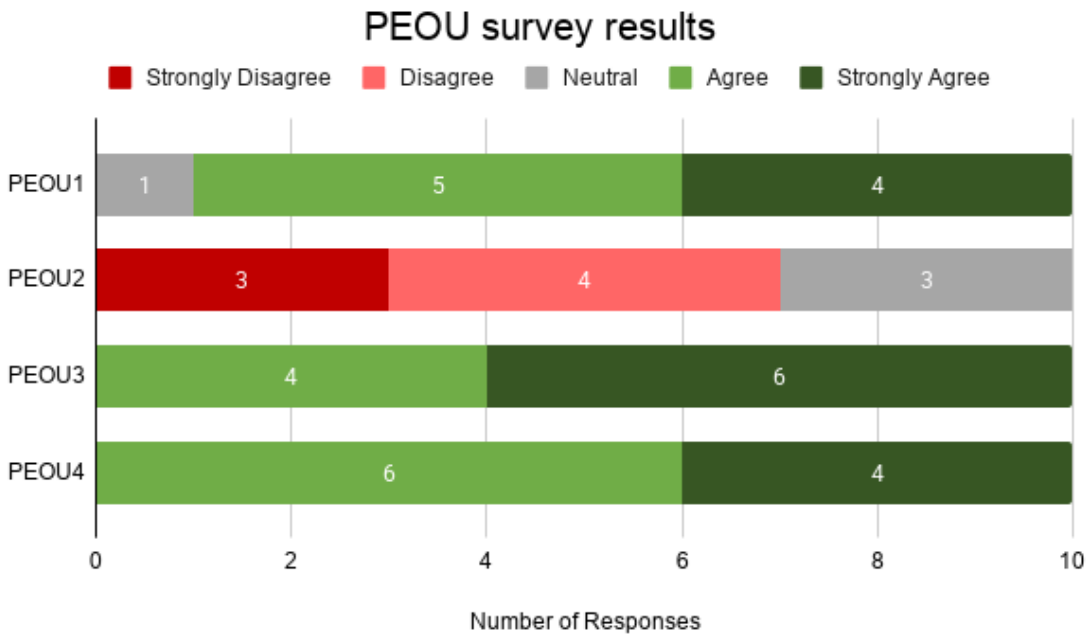


Figure G.1: PEOU survey results, showing the distribution of responses across the five-point Likert scale for each PEOU question.
Note: PEOU1: The interaction with the tool was clear and understandable. PEOU2: I found it difficult to interpret the feedback provided by the tool. PEOU3: The tool's output was presented in a way that was easy to follow. PEOU4: Overall, I found the tool easy to use.

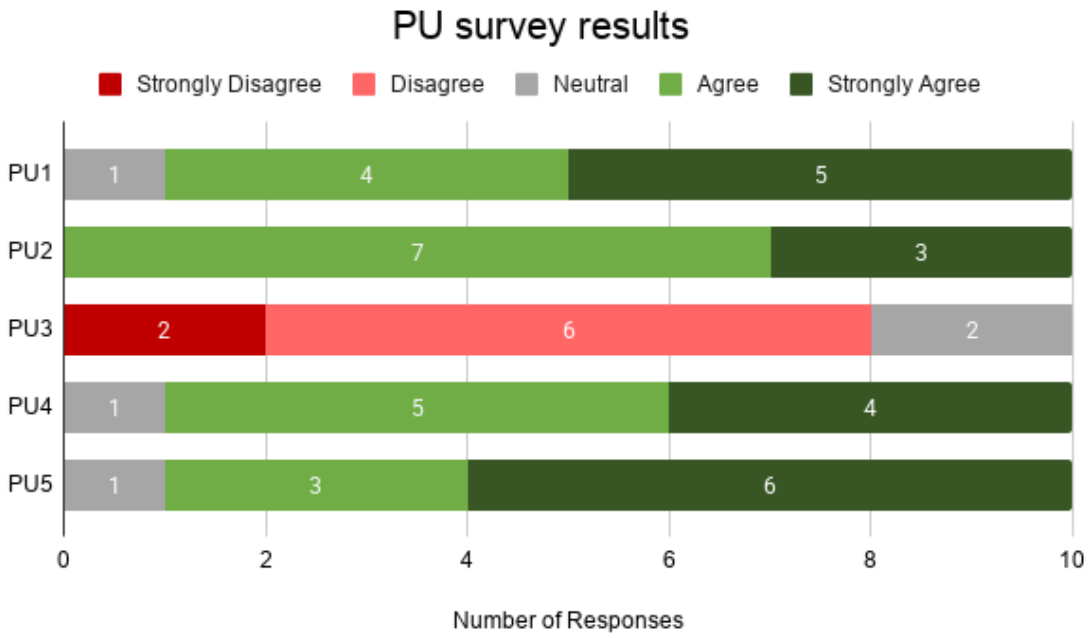


Figure G.2: PU survey results, showing the distribution of responses across the five-point Likert scale for each PU question.

Note: PU1: The tool helped me identify issues in the API specification that I might have missed manually. PU2: Using the tool improved my efficiency in reviewing API specifications. PU3: The tool did not reduce the effort required to validate API specifications. PU4: The tool improved the overall API specification validation process. PU5: Overall, I found the tool useful.

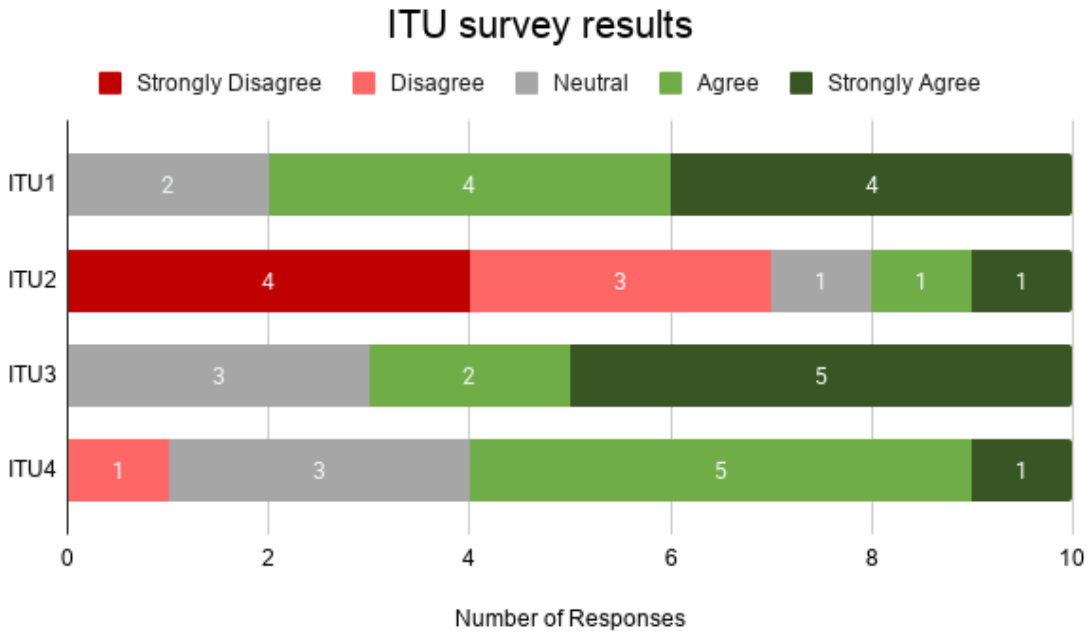


Figure G.3: ITU survey results, showing the distribution of responses across the five-point Likert scale for each ITU question.

Note: ITU1: I would use this tool in my API review workflow. ITU2: I would not recommend this tool to colleagues involved in API development. ITU3: I would be interested in using future versions of this tool. ITU4: I would search for a similar tool if this one became unavailable.