Linguistic Patterns and Antipatterns Detection and their Impact on Understandability and Readability of APIs

by

Krishno Dey

BSc in Computer Science and Engineering, DIU, Bangladesh, 2021

A Thesis Submitted in Partial Fulfilment of
the Requirements for the Degree of

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor: Francis Palma, Ph.D., Faculty of Computer Science
Hung Cao, Ph.D., Faculty of Computer Science

Examining Board: Saqib Hakak, Ph.D., Faculty of Computer Science, Chair
Zahra Khatami, Ph.D., Department of Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

April, 2025

© Krishno Dey, 2025

Abstract

Application Programming Interfaces (APIs) allow distributed systems to expose their functionalities. Despite well-known API design rules and guidelines (patterns), many poor design practices (antipatterns) are prevalent in APIs. This thesis aims to (1) assess the linguistic design quality of APIs and (2) evaluate the impact of patterns and antipatterns on the understandability and readability of APIs through a survey of API developers. We rely on syntactic and semantic analyses to automatically assess the design quality of APIs. Syntactic analysis involves analyzing the structure and syntax of the APIs, while semantic analysis involves analyzing API documentation, descriptions, and parameters. We found that linguistic antipatterns are prevalent in APIs. Our detection algorithms achieve an average detection accuracy of 94%. The survey confirms that adherence to linguistic patterns significantly enhances the understandability and readability of APIs. Our findings will assist API developers in improving the design quality of their APIs.

Dedication

"To my parents, whose constant support and encouragement have been the foundation of my journey. To my sisters, whose kindness and encouragement have been invaluable. To my friends, whose encouragement has kept me motivated. Your belief in me has driven my determination to complete this thesis and reach my goals."

Acknowledgements

I am truly thankful to my supervisors, Dr. Francis Palma and Dr. Hung Cao, whose invaluable guidance and expertise have been instrumental throughout this process. Their mentorship has helped me shape my ideas and improve my work, and I am sincerely grateful for that. Thank you, Dr Palma and Dr Cao, for being my continuous sources of strength and motivation.

We thank the three professionals who participated in the validation/ground truth definition process. The validation/ground truth was compared with the result of the detection tool to generate performance measures such as accuracy, precision, recall, and F1 scores. We also thank professionals for their valuable feedback on our impact survey design. We also extend our gratitude to all the participants who took the time to complete the survey. Finally, we sincerely appreciate the funding and necessary support from the Faculty of Computer Science (FCS), University of New Brunswick, for conducting this thesis.

This research was supported by the New Brunswick Innovation Foundation (NBIF) through NBIF TRF award #TRF2023-003. We acknowledge the financial and institutional support from the Faculty of Computer Science, University of New Brunswick.

Table of Contents

Abstract ii

Dedication iii

Table of Contents v

List of Tables viii

List of Figures xi

1 Introduction 1

1.1 Research Questions 4

1.2 Main Contributions 4

1.3 Findings 5

1.4 Outline 5

2 Background 6

2.1 Service-Based Software Systems 6

2.2 Microservices 7

2.3 Web APIs 8

2.4 API Design Quality 9

3 Related Work 10

3.1 Linguistic Design Quality of Interfaces in Object Oriented Systems . 10

3.2 Linguistic Design Quality of Web Services Interfaces 12

3.3 Linguistic Design Quality of APIs 13
3.4 Impact of Poor Linguistic Design Quality on Maintainability 16

4 Linguistic Design Patterns and Antipatterns in APIs **20**
4.1 ✗*Amorphous* vs. ✓*Tidy Endpoint* 20
4.2 ✗*Contextless* vs. ✓*Contextualized Resource Names* 22
4.3 ✗*CRUDy* vs. ✓*Verbless Endpoint* 23
4.4 ✗*Inconsistent* vs ✓*Consistent Documentation* 24
4.5 ✗*Non-descriptive* vs. ✗*Descriptive Endpoint* 25
4.6 ✗*Non-hierarchical* vs ✓*Hierarchical Nodes* 26
4.7 ✗*Non-pertinent* vs ✓*Pertinent Documentation* 27
4.8 ✗*Non-standard* vs ✓*Standard Endpoint* 28
4.9 ✗*Pluralized* vs ✓*Singularized Nodes* 29
4.10 ✗*Unversioned* vs ✓*Versioned Endpoint* 30
4.11 ✗*Parameters Tunneling* vs ✓*Adherence* 31
4.12 ✗*Inconsistent* vs ✓*Consistent Resource Archetype Names* 32
4.13 ✗*Identifier Ambiguity* vs ✓*Identifier Annotation* 33
4.14 ✗*Flat* vs ✓*Structured Endpoint* 34

5 Research Design **35**
5.1 Phase 1: Data Collection 36
5.2 Phase 2: Empirical Study 36
5.2.1 Detection of Patterns and Antipatterns 36
5.2.2 Impact Study 38

6 Implementation **41**
6.1 Detection of Linguistic Patterns and Antipatterns 41

6.2 Impact Study 45

7 Results 54

7.1 RQ1: Prevalence of Linguistic Patterns and Antipatterns in APIs . . 54

7.1.1 Discussions on Detection 66

7.1.2 Implications for Developers 70

7.2 RQ2: Impact the Understandability and Readability 72

7.2.1 Discussions on Impact 90

7.2.2 Implications for Developers 92

8 Threats to Validity 93

8.1 External validity 93

8.2 Internal validity 94

8.3 Construct validity 95

8.4 Reliability Validity 96

9 Conclusion and Future Works 98

Bibliography 111

A Impact Study Terminology 112

A.1 Terminology 112

A.2 Best Practices in API Design 114

B Impact Study Statistics 116

Vita

List of Tables

3.1 Comparison with the relevant studies on detecting linguistic patterns and antipatterns in the literature. 17
3.2 Comparison with the relevant studies on the impact of API design practices on their understandability and readability in the literature. 18

5.1 List of 69 analyzed APIs, online documentation, and number of endpoints. 37
5.2 Overview of impact study. 39

6.1 Null Hypotheses with Their Alternatives for the Four Confirmatory RQs. 50
6.2 Effect Size Interpretation Based on Cohen's *d*. 53

7.1 Detection Results on 4,027 Endpoints from 37 REST APIs for 14 Patterns and Antipatterns. 59
7.2 Detection Results on 4,027 Endpoints from 32 GraphQL APIs for 14 Patterns and Antipatterns. 60
7.3 Performance of the detection algorithms. P: Positive, N: Negative, Pre: Precision, Rec: Recall, F1: F1 Score. 64
7.4 Comparison with state-of-the-art detection methods. 71
7.5 Descriptive statistics for RQ2.2, perceived difficulty rating in understandability of APIs ranging from 1 (very easy) to 5 (very difficult), mean TAU provided for comparison. 81

7.6 Descriptive statistics for RQ2.3, perceived difficulty rating in readability of APIs ranging from 1 (very easy) to 5 (very difficult), mean *TAU* provided for comparison. 85

B.1 Descriptive Statistics for RQ2.1. Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern. P: Linguistic design pattern, AP: Linguistic design antipattern. 116

B.2 Extended Descriptive Statistics for TAU (RQ2.1). Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern. P: Linguistic design pattern, AP: Linguistic design antipattern. 117

B.3 RQ2.1 Hypothesis testing for TAU, Holm-Bonferroni adjusted p-values with a significance level of $\alpha = 0.05$. The results are sorted by effect size (Cohen's d) to highlight the most impactful findings. Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern. 117

B.4 RQ2.2 Hypothesis testing for TAU, Holm-Bonferroni adjusted p-values with a significance level of $\alpha = 0.05$. The results are sorted by effect size (Cohen's d). Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern. 118

B.5 Correlation between perceived difficulty in understandability and TAU for RQ2.2. Values are sorted by Kendell's τ (correlation strength), Holm-Bonferroni adjusted p-values, $\alpha = 0.05$. Insignificant correlations are marked with (*). 118

B.6 RQ2.3 Hypothesis testing for TAU, Holm-Bonferroni adjusted p-values with a significance level of $\alpha = 0.05$. The results are sorted by effect size (Cohen's d). Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern. 119
B.7 Correlation between perceived difficulty in understandability and TAU for RQ2.3. Values are sorted by Kendell's τ (correlation strength), Holm-Bonferroni adjusted p-values, $\alpha = 0.05$. Insignificant correlations are marked with (*). 119

List of Figures

2.1 Overview of the Microservice Architecture. 7

5.1 Overview of the research design. 35

5.2 Overview of the linguistic patterns and antipatterns detection methods. 38

5.3 Overview of the impact study. 39

6.1 Pre-processing workflow (Step 1 from Figure 5.2). 42

6.2 Example of a comprehension question based on ✓*Contextualized Resource Names* (left) and ✗*Contextless Resource Names* (right), with the correct answer marked. 46

7.1 Detection of Patterns and Antipatterns in REST APIs. The black portion represents antipatterns and the white portion represents patterns. 55

7.2 Detection of Patterns and Antipatterns in GraphQL APIs. The black portion represents antipatterns and the white portion represents patterns. 56

7.3 Prevalence of antipatterns in REST and GraphQL APIs. 65

7.4 Distribution of the experience of the participants. 72

7.5 Distribution of the profession of the participants. 73

7.6 Distribution of the country of the participants. 73

7.7 Comparison of correctness between the linguistic design patterns and antipatterns (higher is better). The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns. 74
7.8 Comparison of duration between the linguistic design patterns and antipatterns (lower is better). The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns. 75
7.9 Comparison of TAU between the linguistic design patterns and antipatterns (higher is better). The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns. 75
7.10 TAU distributions for ✓*Tidy Endpoint*, ✓*Contextualized Resource Names*, ✓*Verbless Endpoint*, ✓*Consistent Documentation*, ✓*Descriptive Endpoint*, ✓*Non-Hierarchical Nodes*, and ✓*Pertinent Documentation* along with their respective linguistic design antipatterns. P: Linguistic design pattern, AP: Linguistic design antipattern. 77
7.11 TAU distributions for ✓*Standard Endpoint*, ✓*Singularized Nodes*, ✓*Versioned Endpoint*, ✓*Parameter Adherence*, ✓*Consistent Resource Archetype Names*, ✓*Parameter Annotation* and *Structured Endpoint* along with their respective linguistic design antipatterns. P: Linguistic design pattern, AP: Linguistic design antipattern. 78
7.12 Effect sizes for differences in TAU between patterns and antipatterns, ranked by Cohen's d. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns. 79
7.13 Bar plots of the perceived difficulty in understandability (RQ2.2) for the ratings 1 (very easy) and 2 (easy), linguistic design patterns ratings on the left, and linguistic design antipatterns ratings are on the right. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns. 81

7.14 Effect sizes for differences in TAU between linguistic design patterns and antipatterns, ranked by Cohen's d for RQ2.2. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns. 82
7.15 Adjusted R^2 values for the regression between TAU and perceived difficulty in understandability ratings for linguistic design antipatterns, ordered by adjusted R^2, higher is better. 82
7.16 Bar plots of the perceived difficulty in readability (RQ2.3) for the ratings 1 (very easy) and 2 (easy), patterns ratings on the left, and antipatterns ratings on the right. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns. 87
7.17 Effect sizes for differences in TAU between patterns and antipatterns, ranked by Cohen's d for RQ2.3. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns. 88
7.18 Adjusted R^2 values for the regression between TAU and perceived difficulty ratings in readability for antipattern for RQ2.3, ordered by adjusted R^2, higher is better. 88

List of Abbreviations

API - Application Programming Interface

REST - Representational State Transfer

HTTP - Hypertext Transfer Protocol

SaaS - Software as a Service

gRPC - Google Remote Procedure Calls

IoT - Internet of Things

OO Systems - Object-oriented Systems

OOP - Object-Oriented Programming

WSDL - Web Service Definition Language

SOAP - Simple Object Access Protocol

OCCI - Open Cloud Computing Interface

NLP - Natural Language Processing

JSON - JavaScript Object Notation

DOLAR - Detection Of Linguistic Antipatterns in REST

SARA - Semantic Analysis of RESTful APIs

RAMA - RESTful API Metric Analyzer

URI - Uniform Resource Identifier

SWRL - Semantic Web Rule Language

SQWRL - Semantic Query-Enhanced Web Rule Language

HATEOAS - Hypermedia As The Engine Of Application State

LLMs - Large Language Models

Chapter 1

Introduction

Nowadays, microservices are a very popular architectural style for building distributed systems [41]. The microservices architectural style structures an application as a collection of independently deployable and loosely coupled services. Application Programming Interfaces (APIs) facilitate communications among the (micro)services. APIs define the protocols, data formats, and operations for communication [36]. Moreover, distributed systems and microservices rely on APIs and their documentation to publish their functionalities. APIs may follow good design practices that exhibit high-quality design, i.e., *design patterns*. On the other hand, APIs may also exhibit poor design practices, i.e., *antipatterns*. Exposing functionalities through APIs also helps to hide internal logic and complexity. APIs are the first point of contact for client developers. Therefore, the design quality of APIs (e.g., understandability, readability) is important for their long-term success and use [27, 36]. The term *understandability* refers to the ease with which one can understand and interpret the functionality and purpose of an API endpoint [14, 10], and *readability* refers to the ease with which one can read and comprehend an API endpoint [14].

The design quality of APIs plays a major role in effective communication among the

services in the distributed systems. A well-designed API (1) provides the required functionality and interfaces for developers that are clear and easy to use, (2) improves developer experience, (3) reduces errors, eases maintenance, boosts integration, and increases easy adoption [36]. A poorly designed API is hard to understand and use, lacks proper documentation, and increases development time for client developers. Automatically detecting poor linguistic design in APIs is crucial for creating user-friendly and efficient APIs in distributed systems and microservices. Adopting good API design practices and identifying poor design early can significantly improve the success of an API.

Practitioners often interpret API design rules and guidelines differently, leading to misconceptions about design quality. The lack of consensus makes it difficult to establish and follow best practices. Several studies have proposed design rules and principles aligned with API constraints (e.g., REST) to provide a common ground for developers [49, 43, 42, 58, 72, 57]. However, their effectiveness in guiding RESTful API design remains largely unexplored.

This thesis assesses the linguistic design quality of the APIs of distributed systems and microservices by automatically detecting linguistic design patterns and antipatterns. We assess how well APIs expose their functionality. This thesis specifically focuses on assessing the linguistic design quality of REST and GraphQL APIs because they are the two most popular API categories in the industry [68]. REST APIs follow REST (Representational State Transfer) principles and are an industry-standard protocol for communication between Web services and their clients using HTTP (Hypertext Transfer Protocol) requests and responses [20]. GraphQL is another API architecture that has recently grown in popularity and offers a more adaptable and effective substitute for conventional REST APIs. GraphQL allows clients to request the precise data they require, potentially lowering the quantity of data carried over the network [23]. Both REST and GraphQL follow a resource-

oriented architecture [23, 36]. Every resource has its unique identifier, URI (Unique Resource Identifier), and a set of actions (HTTP methods) the client can perform to manipulate resources.

We also assess the impact of linguistic design patterns and antipatterns on the understandability and readability of APIs in distributed systems and microservices. Several studies have analyzed how well real-world APIs comply with REST design constraints [40, 58, 56]. Their findings suggest that few APIs fully adhere to REST constraints in practice. Developers assign different levels of importance to various design rules [8, 58]. Bogner et al. [8] conducted a study in which industry practitioners rated the importance of 82 REST design rules defined by Masse [36]. Their results revealed that only 45 out of 82 rules were rated as having medium or high importance, with maintainability and usability emerging as the most critical design attributes. Maintainability and usability are closely related to understandability and readability [14]. The authors also investigated the impact of REST design rules on API understandability [10]. Their findings indicate that violating REST design rules negatively affects understandability. However, their study did not address readability, another key quality attribute in API design.

This thesis investigates linguistic antipatterns in APIs of distributed systems and microservices, aiming to identify the most prevalent instances and assess their impact. Specifically, we analyze ten linguistic design patterns and antipatterns established in prior research [43, 42, 45], along with four newly defined linguistic patterns and antipatterns, to determine their influence on API understandability and readability. The findings of this thesis will provide empirical evidence of how poor linguistic design (antipatterns) negatively affects API usability. These insights will help API developers identify and address common antipatterns, leading to better-designed and more maintainable APIs.

1.1 Research Questions

This thesis aims to answer the following two main research questions:

RQ1: To what extent can we accurately show the prevalence of linguistic antipatterns in APIs of distributed systems and microservices?

RQ2: To what extent do linguistic patterns and antipatterns impact the understandability and readability of APIs?

1.2 Main Contributions

As our main contributions, this thesis:

- Assesses the linguistic design quality of APIs of distributed systems and microservices using syntactic and semantic analyses;
- Analyzes 4,027 endpoints from 37 REST and 32 GraphQL APIs to identify the most prevalent linguistic patterns and antipatterns;
- Provides empirical evidence that both REST and GraphQL APIs exhibit poor linguistic design and identifies the most occurring patterns and antipatterns among the analyzed APIs;
- Provides empirical evidence that poor design practices (linguistic design antipatterns) have a negative impact on the understandability and readability of APIs.
- Validates this impact through a controlled experiment with 108 participants, evaluating 28 API snippets (14 following linguistic design patterns, 14 exhibiting linguistic antipatterns).

1.3 Findings

Our linguistic pattern and antipattern detection method achieved an average accuracy of 94%. The results indicate that linguistic antipatterns are prevalent in APIs of distributed systems and microservices. The most frequently occurring linguistic antipatterns are ✗ *Unversioned Endpoints*, ✗ *Pluralized Nodes*, and ✗ *Inconsistent Resource Archetype Names*.

Our impact study confirms that linguistic antipatterns negatively affect API understandability and readability. APIs that follow linguistic design patterns are easier to understand and use, while APIs exhibiting linguistic antipatterns are more difficult to interpret and navigate.

1.4 Outline

This thesis is organized as follows: Chapter 2 provides relevant background to our thesis. Chapter 3 highlights the relevant studies from the literature on detecting linguistic design patterns and antipatterns in APIs and their impact on the understandability and readability of APIs. Chapter 4 presents definitions of 14 linguistic antipatterns we are studying. Chapter 5 outlines our research design, while Chapter 6 presents the implementation details. Chapter 7 summarizes the detection results and answers two research questions. Chapter 8 outlines the measures taken to address various threats to the validity. Finally, Chapter 9 concludes our discussion and proposes future work.

Chapter 2

Background

This chapter provides a brief overview of the background relevant to our work. Figure 2.1 illustrates the architecture of service-based systems that integrate microservices, highlighting their modular structure and interactions. We begin by introducing service-based software systems, followed by a concise discussion on microservices, Web APIs, and API design quality. Finally, we summarize existing linguistic design patterns and antipatterns in API design.

2.1 Service-Based Software Systems

Service-based software systems are software applications designed to deliver specific functionalities or services to users [7, 76]. The changing nature of the resources and all other entities make the service-based software systems dynamic [76]. These dynamic service-based systems allow providers to offer different services to clients over the Internet. Services are provided through a modular architecture where individual components act as distinct services that can be accessed and combined as needed. Cloud-based delivery models like Software as a Service (SaaS) are often used to offer

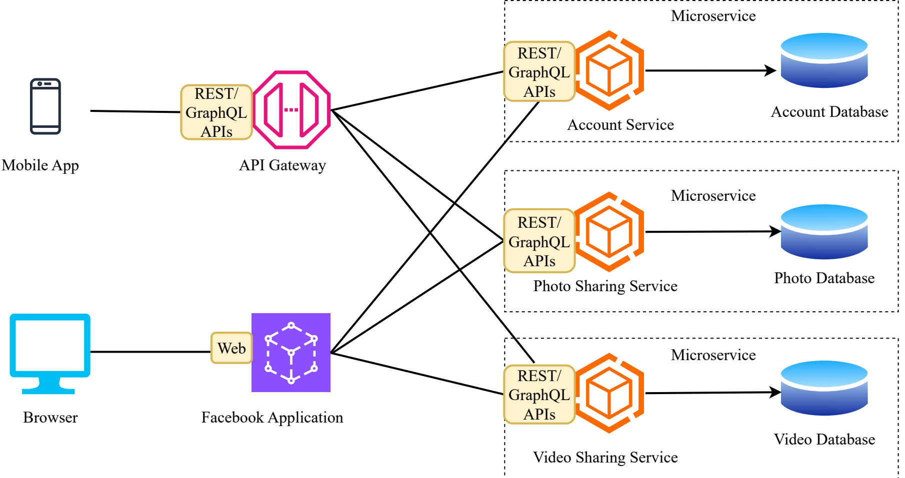

Figure 2.1: Overview of the Microservice Architecture.

services to users. These systems create and deliver value between the provider and the customer through services. Examples of service-based software systems include YouTube, Facebook, Airbnb, Uber, Amazon, etc. [7, 76].

2.2 Microservices

In a microservices architecture, small services (a.k.a. microservices) perform specific tasks and can be independently deployed, scaled, and tested without affecting the entire service-based system [41, 16]. Therefore, microservices allow a flexible development environment for client application developers to develop and deploy their systems into several independent units [41]. Microservices provide numerous benefits such as heterogeneity, resilience, scaling, easy deployment, and many more. Microservices expose their functionalities to client developers via APIs. Communication among these microservices occurs via well-defined APIs over the Internet, which helps enforce separation between the services and avoid dependence [41].

2.3 Web APIs

Web APIs are the backbone of modern web applications [36, 68]. They enable seamless communication and information exchange between different software systems. These interfaces allow applications to request and receive data, invoke operations, and interact with external services without being exposed to their internal complexity. Web APIs enhance modularity, scalability, and interoperability by abstracting complex functionalities, which makes APIs essential in building modern web-based applications.

With the rise of microservices and distributed systems, Web APIs have evolved to meet diverse demands in performance, flexibility, and data transfer efficiency [16]. Client developers rely on API specifications to define operations and structure requests and responses, enabling consistency and interoperability. Well-designed APIs accelerate development cycles and enhance the experience for client developers. However, exposing functionalities through APIs introduces security risks. Developers often implement authentication mechanisms such as OAuth and API keys to protect exposed resources and mitigate threats [66]. Thus, ensuring API security is critical to maintaining data integrity and preventing unauthorized access.

REST (Representational State Transfer), GraphQL, and gRPC (Google Remote Procedure Calls) are three popular paradigms for designing Web APIs [36, 68]. All three types of APIs are tailored to specific use cases and architectural requirements. REST is known for its simplicity and widespread adoption. It relies on standard HTTP methods to perform different operations on resources [36, 72]. In REST, resources are structured as unique URIs and rely on stateless interactions. This makes REST ideal for web and mobile applications that require scalable performance. However, REST APIs often result in over-fetching or under-fetching data in scenarios requiring complex queries. In contrast, GraphQL provides a more flexible query language that

allows clients to request only the data they need [23, 53], which helps to minimize data transfer and improve performance. GraphQL simplifies API interactions by enabling developers to consolidate several REST-style endpoints into a single entry point. On the other hand, the high-performance protocol-buffer-based framework gRPC is well-suited for inter-service communication in distributed systems [71]. It uses HTTP/2 for transport and enables multiplexing and efficient data streaming. The gRPC APIs are particularly suited for real-time applications such as chat-bots and Internet of Things (IoT) applications [71].

GraphQL excels in data-rich contexts, allowing flexible querying; REST prioritizes simplicity and ubiquity, while gRPC provides unparalleled efficiency and performance for microservices and high-throughput applications. Each type of API has its strengths and weaknesses. The selection of API type depends on the specific requirements of the applications and systems.

2.4 API Design Quality

Design quality in APIs refers to the clarity and consistency of their design and documentation for clients [36]. Good API design quality is essential for seamless integration and widespread adoption [36, 68]. A well-designed API is intuitive and easy to comprehend, which leads to faster development cycles, fewer errors, and higher developer satisfaction [72]. In contrast, poorly designed APIs are hard to understand and misleading for the client developers [36, 68]. APIs that follow good design practices typically exhibit *linguistic design patterns* [43, 9], while those that do not, tend to contain *linguistic design antipatterns* [43, 9]. Automatic detection of these poor practices can help client developers improve the overall design quality of their APIs. Despite the availability of several API design guidelines in the literature, poor design quality is prevalent in real-world APIs [59, 43, 42].

Chapter 3

Related Work

This chapter summarizes the current state-of-the-art research relevant to our work. First, we review studies on assessing the linguistic design quality of interfaces in object-oriented systems. Next, we discuss existing research on evaluating the linguistic design quality of APIs. Finally, we summarize studies on the impact of API design practices on API understandability and readability.

3.1 Linguistic Design Quality of Interfaces in Object Oriented Systems

Object-oriented (OO) systems are software systems designed using the principles of object-oriented programming (OOP) [2]. Object-oriented systems help encapsulate data and enable modular, reusable, and scalable software design. Several works in the literature use semantic and syntactic analysis to detect poor design quality (bad practices) in OO interfaces. Abebe et al. [1] evaluated the linguistic design quality of source code in OO systems by employing semantic analysis. Their study aimed to improve the linguistic quality of OO source code. Good-quality and self-descriptive

source code and comments are useful in developing highly maintainable systems. Khamis et al. [31] utilized heuristic-based rules to assess the in-line source code documentation in OO systems. The authors aimed to evaluate linguistic quality and consistency between source code and comments.

Other studies in the literature also analyze different phases of the software development life-cycle by employing semantic and syntactic analysis [5, 35, 54]. For example, Lu et al. [35] proposed a method to improve code searches by identifying relevant synonyms using the WordNet English lexical database [19]. Arnaoudova et al. [5] analyzed the renaming of identifiers in OO systems. Finally, Rahman et al. [54] proposed a technique for automatically suggesting relevant search terms derived from the textual descriptions of change tasks in software. These approaches are tailored to assess the linguistic design quality of several artifacts of OO systems [4, 1] or to traditional SOAP-based web services interfaces [37, 59].

Arnaoudova et al. [4] defined 17 linguistic antipatterns for OO programming. The authors implemented the detection algorithms for the proposed linguistic antipatterns. The authors used heuristic-based rules and performed semantic analysis to identify these linguistic antipatterns. Later, other researchers also applied machine learning techniques to detect these linguistic antipatterns in OO source code [17]. Fakhoury et al. [17] performed a comparative study to explore how conventional machine learning methods perform compared to deep learning methods. Aghajani et al. [3] conducted a study to evaluate whether client developers are more likely to introduce bugs when working with APIs that exhibit linguistic antipatterns. The authors conducted their study on more than 1.5k releases of 75 Maven libraries, 14k open-source Java projects based on those libraries, and more than 4k questions related to the libraries from Stack Overflow. The results of their study indicated that the presence of linguistic antipatterns influences the number of bugs introduced. However, these studies were conducted on linguistic antipatterns in OO systems. OO systems

differ from APIs in many aspects, such as design, organization and etc. Therefore, they cannot be applied to assess the linguistic design quality of Web APIs.

3.2 Linguistic Design Quality of Web Services Interfaces

Assessment of linguistic design quality is not only limited to source code in OO systems. Several studies in the literature have also analyzed the design quality and development of web services [37, 59]. In an investigative study, Rodriguez et al. [59] identified linguistic practices on a set of WSDL (Web Service Definition Language) descriptions. These antipatterns focus on the comments, element names, or types used to represent the data models in WSDL documents. Later, Mateos et al. [37] proposed a detection tool to detect a subset of the antipatterns.

Wei et al. [73] proposed a framework and algorithms to analyze service interfaces in SOAP-based web services. In order to facilitate their integration and interoperability, the authors specifically targeted large, overloaded services. The presented framework enabled the refactoring of large interfaces. The performance of the framework was validated with real commercial logistic systems like FedEx. Bertolino et al. [6] designed the behavior protocol of SOAP web services, outlining how clients should interact with the service. The authors introduced the StrawBerry method, which automatically derives the web service behavior protocol from its WSDL interface.

APIs and SOAP-based web services both enable software systems to communicate, but APIs offer broader flexibility beyond just web-based interactions. They also differ in many other aspects, such as communication protocols, development paradigms, and technologies. Therefore, detection methods for SOAP-based web services cannot be applied to detect linguistic design patterns and antipatterns in APIs.

3.3 Linguistic Design Quality of APIs

Several studies in the literature have investigated the linguistic design quality of APIs. This thesis focuses on works that analyze API interfaces, their descriptions, and HTTP message exchanges to evaluate the language design quality of APIs.

Books in the existing literature have explored the importance of REST API design principles and their impact on usability, readability, and maintainability [36, 68, 57]. These books highlight the importance of adhering to linguistic design patterns (good practices) to ensure consistency and ease of use in APIs. The authors emphasize the need for good naming conventions for resources, proper use of HTTP methods, proper documentation of resources, and standardization of response structures. Moreover, the authors also state that APIs that follow standard good API design principles are easier for developers to understand and use [36, 72]. In contrast, poorly designed APIs are hard to understand and are misleading [68, 57]. Therefore, it is necessary to study the linguistic design quality of the APIs to identify good and poor design practices. Identifying these poor practices in API design will help API developers improve the overall design quality of their APIs.

Researchers assessed the linguistic design quality of real-world APIs. Petrillo et al. [52] surveyed linguistic design quality APIs and presented 73 best practices in REST API design to increase their understandability and reusability. The authors assessed three well-known APIs from three Cloud providers, i.e., Google Cloud Platform, OpenStack, and Open Cloud Computing Interface (OCCI), to evaluate their quality based on the identified best practices. In another work [50], the authors proposed CLOUDLEX and studied the presence of linguistic patterns and antipatterns in 16 cloud computing APIs. Cloud APIs often utilize inconsistent terminology in their URI designs, with more than half of the URIs lacking proper documentation. The CLOUDLEX approach demonstrated an average precision of 85% and a recall of

64%. Brabra et al. [12, 11] defined OCCI REST patterns and antipatterns and presented their detection methods. The authors performed their analysis on six APIs for Cloud services and detection of 28 OCCI REST antipatterns. Hausenblas [26] proposed a method to detect bad practices in REST API design to assess the quality of endpoint naming by performing a subjective analysis on REST APIs. Similarly, Parrish [48] utilized subjective lexical comparison to analyze the use of verbs and nouns in endpoint naming.

Haupt et al. [24] presented a framework for structural analysis of APIs based on their documentation. The presented framework was focused on analyzing the structural properties of APIs. The authors later extended the study to API governance [25]. Panziera et al. [47] proposed a set of best practices for developing self-descriptive REST services that are both human-readable and machine-processable, such as utilizing a common vocabulary for REST resources. The authors introduced a framework for gathering documentation information to generate descriptions of REST services. Upon evaluation, their framework achieved 72% precision and 77% recall in correctly identifying resources. Treude et al. [69] developed a search-based approach for automatically extracting tasks (i.e., specific programming actions to be undertaken) from software documentation. The authors aimed to minimize the gap between the information needs of the developers and the documentation structure/content, thus assisting developers in documentation navigation. The suggested approach utilizes natural language processing (NLP) techniques and was able to extract more than 70% of tasks from two large corpora of software documentation.

Neumann et al. [40] analyzed 500 APIs to evaluate key technical features, compliance with REST architectural principles and adherence to best design practices. The authors identified several trends, such as widespread support for JavaScript Object Notation (JSON) and the use of software-generated documentation. However, the authors also observed significant diversity in services, including varying degrees of

adherence to best practices, with only 80% percent of the APIs strictly complying with all REST principles. Rodriguez et al. [58] analyzed 78 GB of HTTP traffic to assess the linguistic design quality of Web APIs. The authors employed 18 REST-aligned heuristics to evaluate their design quality and found that most of the APIs are on level 2 of the Richardson maturity model [21], and a few APIs are on level 3. Palma et al. [43, 42, 45, 44, 46] have conducted a series of studies to evaluate linguistic design patterns (good API design practices) and antipatterns (poor API design practices) in real-world APIs. The authors proposed automatic detection tools such as DOLAR [43] and SARA [42] to perform syntactic and semantic analysis on APIs. Moreover, the authors defined nine patterns and antipatterns and conducted their study on widely known REST APIs. Later, the authors studied APIs of IoT applications [45], analyzed 1,102 endpoints from 19 IoT APIs, and detected nine linguistic patterns and antipatterns. In another work, the authors defined another linguistic design pattern and antipattern and conducted a comparative study on 37 public, partner, and private APIs [44]. The main objective of the study was to investigate which type of API is more prone to poor linguistic design. Bogner et al. [9] also developed a similar automatic detection tool named RESTRuler. RESTRuler employs heuristics-based rules for detecting compliance with REST rules and principles in real-world APIs. Other studies were carried out using the detection methodologies developed by Palma et al. [42] to detect antipatterns in different types of APIs such as Cloud APIs [50] and Google APIs [46].

Several surveys on API design quality provided a state-of-the-art summary of the current good and bad practices in API design [22, 32]. Their result also showed the existence of poor design practices (linguistic design antipattern) in real-world APIs. Moreover, their finding also indicated that the presence of poor design practices hinders the understandability and readability of APIs. Bogner et al. [8] also conducted an empirical study to investigate how API design rules influence the understand-

ability of APIs. Their finding suggested that adherence to good design principles positively impacts comprehension. Table 3.1 summarizes the state-of-the-art methods for detecting linguistic design patterns and antipatterns in APIs.

3.4 Impact of Poor Linguistic Design Quality on Maintainability

The poor linguistic design quality of APIs impacts their maintainability (understandability and readability) [36, 10]. Readability refers to the ease with which one can read and comprehend an API endpoint [14]. It involves clarity, organization, and the use of intuitive naming conventions. A readable API enables developers to quickly grasp its design without unnecessary cognitive load. Similarly, the term understandability refers to the ease with which one can understand and interpret the functionality and purpose of an API endpoint [14, 10]. Understandability extends beyond just reading the API syntax. It also includes how well the behavior, usage, and purpose of APIs are conveyed, ensuring developers can integrate and utilize them effectively in their systems. In the following, we summarize state-of-the-art techniques to assess the impact of poor linguistic design quality on their maintainability.

Haupt et al. [25] focused on the governance of RESTful APIs, utilizing the structural API analysis framework [24] the authors previously developed to conduct a structural analysis of 286 real-world Web APIs. The proposed framework processes API documentation and converts it into a canonical metamodel. Then, it calculates various metrics to assess the API governance. The authors also demonstrated how these metrics could estimate user-perceived complexity, a concept closely tied to API understandability and readability. However, a larger-scale confirmatory study is needed to validate such findings and claims. Bogner et al. [8] used similar techniques

Table 3.1: Comparison with the relevant studies on detecting linguistic patterns and antipatterns in the literature.

Study	Year	Goal of the Study	Target APIs	Method/Technique
Parrish [48]	2010	Lexical analysis of 2 social network APIs	Facebook and X (formerly Twitter)	Subjective lexical comparison for analyzing the use of verbs and nouns in endpoint naming
Wei et al. [73]	2015	Structural analysis of the service interfaces	272 operations from 13 cloud services related to Amazon, FedEx, etc.	Algorithmic rules to analyze Web service interfaces
Palma et al. [43]	2015	Detection of linguistic antipatterns in APIs	15 public RESTful APIs	Patterns and antipatterns detection tool named DOLAR (Detection Of Linguistic Antipatterns in REST).
Brabra et al. [12]	2016	Definition and detection of OCCI REST patterns and antipatterns	6 APIs for Cloud services	A semantic-based approach that utilizes SWRL (Semantic Web Rule Language) rules and SQWRL (Semantic Query-Enhanced Web Rule Language) queries to define and detect antipatterns.
Petrillo et al. [52]	2016	Check APIs for conformance to 73 best practices	3 Cloud APIs (Google Cloud Platform, OpenStack, and OCCI)	Manual assessment of the conformance to best practices in cloud APIs at the service level
Rodriguez et al. [58]	2016	Check APIs for compliance or violation to 6 standardized practices	78 GB of HTTP traffic from Telecom Italia	A manual evaluation of HTTP methods to ensure compliance with standardized semantics and the structural design of request endpoints.
Haupt [24]	2017	Structural analysis of APIs	286 Swagger API description documents	A canonical metamodel-based framework for structural analysis of REST APIs
Palma et al. [42]	2017	Detection of linguistic antipatterns in APIs	18 APIs for Web applications	SARA (Semantic Analysis of RESTful APIs), an improvement version of DOLAR
Petrillo et al. [50]	2018	Linguistic quality assessment of Cloud Computing APIs	23,062 URIs from the 16 Cloud API providers	CLOUDLEX tool to detect linguistic patterns and antipatterns
Brabra et al. [11]	2019	Detection of OCCI REST patterns and antipatterns	5 Cloud APIs including OCCI, CDAPS, OpenNebula, Amazon S3, and Rackspace	Semantic-based analysis for detecting REST and OCCI antipatterns
Palma et al. [45]	2022	Detection of linguistic antipatterns and patterns in APIs from IoT providers	19 APIs from 18 IoT providers	SARAv2, an improvement version of SARA
Bogner et al. [9]	2024	Detection of linguistic REST design rules	2,300 public OpenAPI descriptions	RESTRuler tool for detection of REST rules and their violation
Dey et al. [15]	2024	Detection of linguistic patterns and antipatterns in REST and GraphQL APIs	21 REST and 12 GraphQL APIs	Semantic and syntactic analysis to detect linguistic patterns and antipatterns
This Work	2025	Proposed four new linguistic patterns and antipatterns and their detection heuristics.	4,027 API endpoints from 69 APIs	Heuristics-based semantic and syntactic analysis to detect linguistic patterns antipatterns

Table 3.2: Comparison with the relevant studies on the impact of API design practices on their understandability and readability in the literature.

Study	Year	Goal of the Study	Target APIs	Method/Technique
Haupt et al. [25]	2018	Assess the API governance	286 REST APIs	Processes API documentation and converts it into a canonical meta-model. Then, it calculates various metrics to assess the API governance
Bogner et al. [8]	2019	Automatic evaluation of Web APIs	1,737 REST APIs	Generates a hierarchical model from API documentation and calculates 10 maintainability metrics
Bogner et al. [10]	2023	Assesses the impact of REST design rules on their understandability	12 REST API snippets following REST rules and 12 REST API snippets violating REST design rules	Use subjective API understandability ratings collected from human participants to investigate how REST API design rules impact the understandability of APIs.
This Study	2025	Assesses the impact of linguistic patterns and antipatterns on their understandability and readability	14 API snippets following linguistic patterns and 14 API snippets following linguistic antipatterns	Collects API understandability and readability ratings from participants to investigate how linguistic patterns and antipatterns impact the understandability and readability of APIs.

as Haupt et al. [24] and proposed a modular framework named RESTful API Metric Analyzer (RAMA). The authors used RAMA to automatically evaluate Web APIs and calculated several metrics for assessing the maintainability of APIs. RAMA framework generates a hierarchical model from API documentation and calculates 10 maintainability metrics. The authors applied RAMA on 1,737 real-world APIs and found poor design practices. The findings also revealed the prevalence of linguistic design antipatterns in real-world APIs. Bogner et al. [10] used subjective API understandability ratings collected from human participants. The authors investigated how REST API design rules impact the understandability of APIs. However, the authors do not assess the impact of design rules on the readability of APIs.

To our knowledge, none of the previous works focuses on the impact of API design practices on the readability of APIs. Most of the works in the literature tried to assess the readability of source codes [62, 29, 18].

In summary, most of the works in the literature focus on assessing the quality of web

APIs. There have been many automatic tools for assessing the quality of APIs (detection of linguistic design patterns and antipatterns), such as DOLAR [43], SARA [42], and RESTRuler [9]. Findings from these studies show the prevalence of poor API design practices in real-world APIs and suggest that poor design practices hinder the understandability and readability of APIs. However, only a few studies evaluate the impact of adhering to or deviating from good API design practices [25, 34, 10]. In this thesis, we aim to assess the linguistic design quality of the APIs. We also aim to evaluate the impact of API design practices (linguistic design patterns and antipatterns) on API understandability and readability.

Chapter 4

Linguistic Design Patterns and Antipatterns in APIs

This chapter presents 14 linguistic design patterns and antipatterns, of which ten are extracted from the literature, and four are newly defined. Throughout the rest of the thesis, we use ✓(green tick) to represent linguistic patterns and ✗(red cross) to indicate linguistic antipatterns.

4.1 ✗*Amorphous* vs. ✓*Tidy Endpoint*

An endpoint is considered ✓*Tidy Endpoint* if it has (1) lower-case resource naming, (2) no underscores, (3) no trailing slashes, and (4) no file extensions. In contrast, ✗*Amorphous Endpoint* antipattern occurs when endpoints have either capital letters (except for camel cases), underscores, trailing slashes, or file extensions that make them difficult to use and read [43].

Example:

- The endpoint `/Available-Data-Feeds/` is an ✗*Amorphous Endpoint* as it contains trailing slashes and uppercase letters.
- In contrast, `/available-data-feeds/{dataSourceId}` is a ✓*Tidy Endpoint* as it does not contain any uppercase letters, underscores, trailing slashes, or file extensions.

Detection Heuristic:

```
1:  AMORPHOUS-ENDPOINT(Request-Endpoint):
2:    if uppercase or underscore or trailing-slash or file-extensions in
Request-Endpoint
3:      return 'Amorphous Endpoint' antipattern
4:    end if
5:    return 'Tidy Endpoint' pattern
```

4.2 ✗*Contextless* vs. ✓*Contextualized Resource Names*

Nodes in the endpoint should belong to the same semantic context, i.e., the endpoints should be contextual. ✗*Contextless Resource Names* antipattern occurs when nodes in the endpoint do not belong to the same semantic context [43]. In the case that nodes in the endpoint belong to the same semantic context, it is known as ✓*Contextualized Resource Names* pattern.

Example:

- The endpoint `/newspapers/earth/players/{id}` is a ✗*Contextless Resource Names* antipattern because nodes are not semantically related.
- In contrast, the endpoint `/football/club/players/{id}` is a ✓*Contextualized Resource Names* pattern because all nodes are semantically related.

Detection Heuristic:

```
1:  CONTEXTLESS-RESOURCE(Request-Endpoint, API-Documentation)
2:    TopicsModel ← Extract-Topics(API-Documentation)
3:    EndpointNodes ← Extract-Endpoint-Nodes(Request-Endpoint)
4:    Similarity-Value ← Calculate-Similarity(EndpointNodes, TopicsModel)
5:    if Similarity-Value < Threshold:
6:      return 'Contextless Resource Names' antipattern
7:    end if
8:    return 'Contextual Resource Names' pattern
```

4.3 ✗*CRUDy* vs. ✓*Verbless Endpoint*

The ✓*Verbless Endpoint* does not use ✗*CRUDy* terms such as create, read, update, delete, or their synonyms. In contrast, the use of such terms as resource names is ✗*CRUDy Endpoint* [43]. Moreover, resources should be identified using nouns, instead of verbs.

Example:

- The endpoint `update/players/{id}` is a ✗*CRUDy Endpoint* antipattern as it has CRUDy term "update".
- In contrast, endpoint `/players/{id}` is a ✓*Verbless Endpoint* pattern as the endpoint does not contain any CRUDy terms or their synonyms.

Detection Heuristic:

```
1:  CRUDY-ENDPOINT(Request-Endpoint):
2:    CRUDyWords ← {"create", "read", "update", "delete", "get", ...}
3:    if CRUDyWords in Request-Endpoint
4:      return 'CRUDy Endpoint' antipattern
5:    end if
6:    return 'Verbless Endpoint' pattern
```

4.4 ✗*Inconsistent* vs ✓*Consistent Documentation*

✗*Inconsistent Documentation* antipattern occurs when the HTTP method (i.e., the action) of an endpoint is in contradiction with its documentation. In contrast, for ✓*Consistent Documentation* pattern, the HTTP method (the action) agrees with the documentation [45].

Example:

- In Adobe Audience Manager API, the HTTP method (POST) of the endpoint `/datasources/bulk-delete` is in contradiction with the documentation 'Bulk delete multiple data sources', and thus is an ✗*Inconsistent Documentation antipattern*. According to the API design guidelines, the POST method should be used to create some resources [36].
- In contrast, in Pipefy API, the HTTP method (POST) of the endpoint `/createCardRelation` is consistent with its documentation 'Creates a card relation,' and thus, is a ✓*Consistent Documentation pattern*.

Detection Heuristic:

```
1:  INCONSISTENT-DOCUMENTATION(HTTP-Method, Request-Endpoint, Documentation)
2:    Documentation ← Remove-Stop-Words(Documentation)
3:    Action ← Extract-Intended-Action(Documentation)
4:    if ((HTTP-Method = 'POST' and Action in SYNONYMS (Delete or Update or Get))
or
5:      (HTTP-Method = 'DELETE' and Action in SYNONYMS (Create or Update or Get))
or
6:      (HTTP-Method = 'PUT' and Action in SYNONYMS (Create or Delete or Get)) or
7:      (HTTP-Method = 'GET' and Action in SYNONYMS (Delete or Update or
Create)))
8:      return "Inconsistent Documentation" antipattern
9:    end if
10:   return "Consistent Documentation" pattern
```

4.5 ✗*Non-descriptive* vs. ✗*Descriptive Endpoint*

In API design, endpoints must be as user-friendly as possible. An endpoint needs to be easy to understand and as precise as possible. When an endpoint design has encoded nodes (e.g., basic resource names not used), it becomes a ✗*Non-descriptive Endpoint* antipattern and gets harder to comprehend. A ✗*Self-descriptive Endpoint*, on the other hand, has resource identifiers that are short and to the point [44].

Example:

- The endpoint `/auth/token/oauth1` is a ✓*Non-descriptive Endpoint* as it is not descriptive enough and hard to understand the purpose of the endpoint.
- In contrast, the endpoint `/account/set-profile-photo` is a ✗*Self-descriptive Endpoint* as the endpoint is descriptive and easy to understand.

Detection Heuristic:

```
1:  NON-DESCRIPTIVE-ENDPOINT(Request-Endpoint)
2:    Nodes ← Split-Compound-Words(Request-Endpoint)
3:    Nodes ← Expand-Acronyms(Nodes)
4:    isValidWord = False
5:    for each word in Nodes:
6:      if PerformWordLookup(word):
7:        isValidWord = isValidWord or True
8:      else
9:        isValidWord = isValidWord or False
10:     if isValidWord:
11:       return 'Descriptive Endpoint' antipattern
12:   end if
13:   return 'Non-Descriptive Endpoint' pattern
```

4.6 ✗*Non-hierarchical* vs ✓*Hierarchical Nodes*

The nodes in endpoints in the ✓*Hierarchical Nodes* pattern are in a hierarchical relationship. In contrast, a ✗*Non-hierarchical Nodes* antipattern occurs when at least one node in an endpoint is not hierarchically related to its neighbor nodes [43].

Example:

- The endpoint `/professors/university/faculty/` is a ✓*Non-hierarchical Nodes* antipattern since 'professors', 'faculty', and 'university' are not in a hierarchical relationship.
- In contrast, `/university/faculty/professors/` is a ✗*Hierarchical Nodes* pattern since 'university', 'faculty', and 'professors' are in a hierarchical relationship.

Detection Heuristic:

```
1:  NON-HIERARCHICAL-NODES(Request-Endpoint)
2:    Nodes ← Extract-Endpoint-Nodes(Request-Endpoint)
3:    for each index in Length(Nodes):
4:      if Is-Hierarchical-Relation(Nodes_index, Nodes_index+1) = false or
5:      Is-Specialisation-Relation(Nodes_index, Nodes_index+1):
6:        return 'Non-Hierarchical Nodes' antipattern
7:      end if
8:    end for
9:    return 'Hierarchical Nodes' pattern
```

4.7 ✗*Non-pertinent* vs ✓*Pertinent Documentation*

✗*Non-pertinent Documentation* antipattern occurs when an endpoint and its corresponding documentation are not semantically related. In contrast, a properly documented endpoint uses semantically related terms to clearly describe its purpose, denoted as ✓*Pertinent Documentation* pattern [42].

Example:

- In PokéAPI, the endpoint `/v2/berry-firmness/{names}/` is not semantically related with its documentation 'The name of this resource is listed in different languages', and thus is a ✗*Non-pertinent Documentation* antipattern.
- In contrast, another endpoint-documentation pair from PokéAPI: `/v2/berry-firmness/{berries}/` – 'A list of the berries with this firmness.' shows a higher semantic relationship, and thus is considered a ✓*Pertinent Documentation* pattern.

Detection Heuristic:

```
1:  NON-PERTINENT-DOCUMENTATION(Request-Endpoint, Documentation)
2:    Documentation ← Remove-Stop-Words(Documentation)
3:    Tokens ← Lemmatise-Tokenise(Documentation)
4:    TopicsModel ← Extract-Topics(API-Documentation)
5:    EndpointNodes ← Extract-Endpoint-Nodes(Request-Endpoint)
6:    Similarity-Value ← Calculate-Similarity-Score(EndpointNodes, TopicsModel)
7:    if Similarity-Value < threshold:
8:      return 'Non-Pertinent Documentation' antipattern
9:    end if
10:    return 'Pertinent Documentation' pattern
```

4.8 ✗*Non-standard* vs ✓*Standard Endpoint*

A ✓*Standard Endpoint* design does not contain (1) non-standard characters such as é, å, ø, etc, (2) blank spaces, (3) unknown characters, and (4) double hyphens. In contrast, The use of characters such as é, å, ø, etc., the presence of blank spaces, the usage of double hyphens, and the presence of unknown characters in the endpoint are the four main indicators of ✗*Non-standard Endpoint* design [45].

Example:

- The endpoint `/data--feeds/billingreport` from IBM Cloud Pak System API is an example of ✗*Non-standard Endpoint* Design as endpoint contains a double hyphen.
- In contrast, the endpoint `/data-feeds/billing-report` represents ✓*Standard Endpoint* design.

Detection Heuristic:

```
1:  NON-STANDARD-ENDPOINT(Request-Endpoint)
2:    if (Non-English-Characters or
      Space or Double-Hyphens or Unknown-Characters) in Request-Endpoint:
3:      return 'Non-standard Endpoint' antipattern
4:    end if
5:    return 'Standard Endpoint' pattern
```

4.9 ✗*Pluralized* vs ✓*Singularized Nodes*

Endpoints should use singular/plural nouns consistently when naming resources in the API. The last node of the request endpoint should be singular when clients send PUT/DELETE requests, thus, a ✓*Singularized Nodes* pattern occurs. In contrast, the last node should be plural for POST requests. Consequently, when singular names are used for POST requests or plural names are used for PUT/DELETE requests, the ✗*Pluralized Nodes* antipattern occurs [43].

Example:

- In Adobe Audience Manager API, the POST method is used with the `/data-feeds/usageendpoint` whose last node is a singular noun, and, thus, it is ✗*Pluralized Nodes* antipattern.
- In contrast, if PUT or DELETE were used with the same endpoint, it would have been ✓*Singularized Nodes* pattern, as singular last nodes are supposed to be used with PUT or DELETE methods.

Detection Heuristic:

```
1:  PLURALIZED-NODES(Request-Endpoint, HTTP-Method)
2:    Last-Node ← Get-Last-Node(Request-Endpoint)
3:    Second-Last-Node ← Get-Second-Last-Node(Request-Endpoint)
4:    if (HTTP-Method = 'PUT' or 'DELETE' and Is-Plural(Last-Node) = true) or
5:      (HTTP-Method = 'POST' or 'DELETE' and Is-Plural(Second-Last-Node) =
false):
6:      return 'Pluralized Nodes' antipattern
7:    end if
8:    return 'Singularized Nodes' antipattern
```

4.10 ✗*Unversioned* vs ✓*Versioned Endpoint*

✓*Versioned Endpoint* makes maintenance simpler for client developers as well as API providers. The format or type of response data may change, a resource may be removed, a new endpoint may be added, response parameters may change, and major or minor API versions are needed to track all the changes. An endpoint exhibits the ✗*Unversioned Endpoint* antipattern if not versioned, while an endpoint using version information as part of its design is known as ✓*Versioned Endpoint* pattern [45].

Example

- For example, the endpoint `/file_requests/count` from Dropbox is an textcolorred✗*Unversioned Endpoint* antipattern because the endpoint does not contain any version information.
- In contrast, another endpoint `/v1/me/library/playlists/{id}` from Apple Music is a ✓*Versioned Endpoint* pattern because the endpoint contains the version number.

Detection Heuristic

```
1:  UNVERSIONED-ENDPOINT(Request-Endpoint):
2:    if Is-Version-Info-Available(Request-Endpoint) = true:
3:      return 'Versioned Endpoint' pattern
4:    end if
5:    return 'Unversioned Endpoint' antipattern
```

4.11 ✗*Parameters Tunneling* vs ✓*Adherence*

Web API clients often required to provide additional information (parameters) via path parameters and query parameters with the endpoint. Query parameters are normally used to sort, filter, and paginate request data, while path parameters are used to identify or retrieve a specific resource [36, 68]. Query parameters can be used with the GET method for sorting, filtering, and paginating resources. Path parameters, on the other hand, can be used with any HTTP method to identify or retrieve a specific resource [36, 68]. Thus, the use of query parameters with methods other than GET results in ✗*Parameter Tunneling* antipattern. Conversely, the consistent use of the correct HTTP method with query and path parameters is considered as ✓*Parameter Adherence* pattern.

Example:

- An endpoint `/api/books?category=fiction` with the PUT method illustrates ✗*Parameter Tunneling* antipattern.
- Another endpoint `/api/books/{id}` with PUT exemplifies ✓*Parameter Adherence* pattern.

Detection Heuristic:

```
1:  PARAMETERS-TUNNELING(Request-Endpoint, HTTP-Method)
2:    HasQueryParam ← Extract-Query-Parameter(Request-Endpoint)
3:    if HasQueryParam = true and HTTP-Method ≠ GET:
4:      return 'Parameter Tunneling' antipattern
5:    else:
6:      return 'Parameter Adherence' pattern
```

4.12 ✗*Inconsistent* vs ✓*Consistent Resource Archetype Names*

Resource archetypes serve as the fundamental building blocks of API endpoints [36]. Four commonly used resource archetypes are Document, Collection, Store, and Controller. Singular resource names should be used for Documents, while plural names for Collections and Stores. Verb phrases should be used for Controllers [36]. Inconsistencies in resource archetype naming, i.e., using singular nouns for Collections or Stores, plural nouns for Documents, or Controllers without the POST method, result in the ✗*Inconsistent Resource Archetype Names* antipattern. In contrast, proper archetype naming results in ✓*Consistent Resource Archetype Names* pattern [9].

Example:

- The endpoint `/recipe/desserts/pie` (following a singular/plural/singular structure) demonstrates the ✓*Consistent Resource Archetype Names* pattern.
- The endpoint `/recipes/desserts/pie` (with a plural/plural structure) exemplifies the ✗*Inconsistent Resource Archetype Names* antipattern.

Detection Heuristic:

```
1:  INCONSISTENT RESOURCE ARCHETYPE NAMES(Request-Endpoint)
2:    Nodes ← Split(Request-Endpoint)
3:    Category ← Categorize-Nodes(Last-Node(Nodes))
4:    if Category = "Document"and Is-Singular(Category) = true:
5:      return 'Consistent Resource Archetype' pattern
6:    else if Category = "Collection"or "Store"and Is-Plural(Category) = true:
7:      return 'Consistent Resource Archetype' pattern
8:    else if Category = "Controller"and Is-Verb-Phrase(Category) = true:
9:      return 'Consistent Resource Archetype' pattern
10:   end if
11:   return 'Inconsistent Resource Archetype' antipattern
```

4.13 ✗*Identifier Ambiguity* vs ✓*Identifier Annotation*

Path parameters or resource identifiers should be enclosed in curly braces, angle brackets, or followed by a colon sign. Using such symbols in endpoint design is referred to as ✓*Identifier Annotation* pattern, which improves endpoint readability and understandability. In contrast, the absence of curly braces, angle brackets, or colon to represent resource identifiers would result in an ✗*Identifier Ambiguity* antipattern.

Example:

- The identifier in endpoint `/v2/lists/:id/members/appInstallation/{id}/rules/` is enclosed in curly braces and follows the ✓*Identifier Annotation* pattern.
- In contrast, the identifier in the endpoint `/rules/ruleKey/idFromLegacyId` is enclosed in any braces making it hard to identify and resulting in an ✗*Identifier Ambiguity* antipattern.

Detection Heuristic:

```
1: IDENTIFIER_AMBIGUITY(Request-Endpoint)
2:   Identifiers ← Extract-Identifiers(Request-Endpoint)
3:   if Identifiers enclosed in "{}" or "<>" or starts with ":":
4:     return 'Identifier Annotation' pattern
5:   end if
6:   return 'Identifier Ambiguity' antipattern
```

4.14 ✗*Flat* vs ✓*Structured Endpoint*

Forward slash (/) must be used to separate nodes of an endpoint and indicate a hierarchical relationship. A ✓*Structured Endpoint* pattern occurs when forward slashes are used to break down complex resource names to improve the readability and understandability of the endpoint. In contrast, a ✗*Flat Endpoint* antipattern occurs when complex or large resource names are not broken down with forward slashes [36].

Example:

- The endpoint `/requests/{request_id}/receipt/requests/{request_id}/map` is an example of a ✓*Structured Endpoint* pattern because it does not have any complex and large resource names.
- In contrast, the endpoint `/requestFirmwareUpdateFromInStoreReader` and `/requestItemDisplayFromInStoreReader` exhibits ✗*Flat Endpoint* antipatterns as it contains complex and large resource names.

Detection Heuristic:

```
1:  FLAT-ENDPOINT(Request-Endpoint)
2:    Nodes ← Split(Request-Endpoint)  on  "/"
3:    for each Node  in  Nodes:
4:      SplittedNodes ← Split(Node)
5:      if SplittedNodes > 2:
6:        return 'Flat Endpoint' antipattern
7:      end if
8:    end for
9:    return 'Structured Endpoint' pattern
```

Chapter 5

Research Design

This chapter describes the research design of this thesis. In Phase 1, we collect API endpoints from real-world APIs. In Phase 2, we conduct two studies: (1) the detection of linguistic patterns and antipatterns in APIs and (2) an impact analysis of linguistic patterns and antipatterns on the understandability and readability of APIs. We describe them in detail in section 5.2. Finally, we use the findings from both studies to answer the defined research questions. Figure 5.1 shows the research design of this thesis.

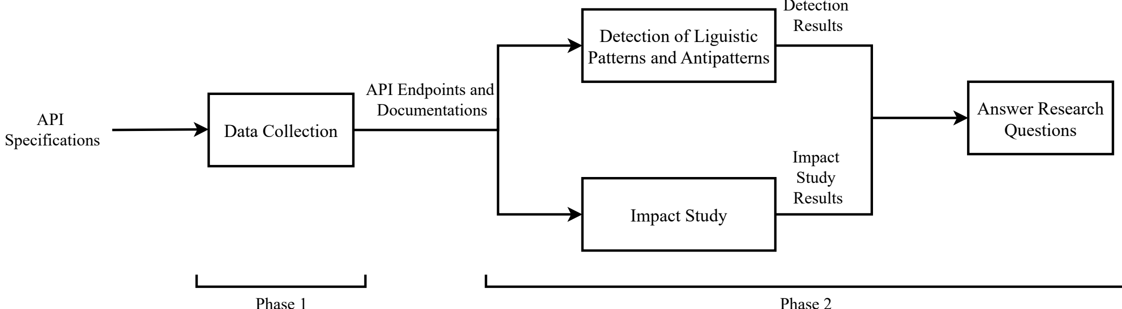

Figure 5.1: Overview of the research design.

5.1 Phase 1: Data Collection

In this phase, we gather information about APIs. We manually collected endpoints, associated HTTP methods, available documentation, and parameters for each API. To ensure the data quality, we followed a systematic data collection approach [38]. We only gathered endpoints with well-organized HTTP methods and documentation for our investigation. The resulting dataset encompasses 1,983 endpoints from 32 GraphQL APIs. Additionally, we incorporate 2,044 endpoints from 37 REST APIs made publicly available by Palma et al. [44] to analyze the design quality of APIs. Table 5.1 lists 69 analyzed REST and GraphQL APIs with the sources of their online documentation and the number of endpoints analyzed.

5.2 Phase 2: Empirical Study

In Phase 2, we perform the detection of linguistic patterns and antipatterns in APIs and conduct the impact survey.

5.2.1 Detection of Patterns and Antipatterns

Figure 5.2 shows the patterns and antipatterns detection methodology followed in this thesis. In Step 1, we pre-process API endpoints and their documentation. We detect linguistic patterns and antipatterns in Step 2 by implementing their detection heuristics. We define the ground truth in Step 3. Finally, we compute detection performance metrics, synthesize our findings, and answer our research questions in Step 4. We provide brief descriptions of each step of the methodology in Section 6.1. To investigate the prevalence of APIs, we further divide RQ1 (as defined in Chapter 1) into four research questions, as outlined below.

Table 5.1: List of 69 analyzed APIs, online documentation, and number of endpoints.

REST		GraphQL	
APIs	Endpoints	APIs	Endpoints
Adobe Audience Manager	65	AniList	27
Amazon AWS Core IoT	150	Apple Music	99
Apple App Store Connect	32	Artsy	21
Arduino IoT Cloud	20	Braintree	96
Bitholic	06	Facebook	66
BroadCom	40	GitHub	256
Cisco Flare	25	GitLab	55
ClearBlade	45	Instagram	28
Coursera	230	Pipefy	90
Dropbox	74	Pokeapi	24
Droplit-io	52	Shopify	33
Google Nest	35	X (formerly Twitter)	51
GroupWise	56	Intuit	41
Guardian	05	Pinterest	92
IBM Watson IoT	57	Hygraph	37
IBM Cloud Pak System	34	Netflix	17
LinkedIn	13	Yelp Fusion	18
LiveAgent	21	NY Times Article Search	11
Losant	63	Swapcard	30
Microsoft Azure IoT Hub	210	ScreenCloud	103
Microsoft Partner Center	70	Satispay	18
Microsoft Power BI	34	Salsify	46
Node-RED	15	Pluralsight	69
OpenAI	67	Northflank	225
Oracle Cloud Marketplace	43	IBM Anomaly Detection	07
Prolateral Core Infrastructure	27	IBM Sterling Supply Chain Insights	194
QuickBooks Online	21	IBM Sterling Inventory Visibility	40
Samsung ARTIK Cloud	80	IBM Partner Marketplace	48
Shopify	71	DailyMotion	56
Sonos	49	Conduit	30
StackOverflow	77	DueDil	33
SurveyJS	24	UberEats	44
The Things Network	11		
X (formerly Twitter)	104		
Uber	14		
WM3 Multishop	54		
YouTube	50		
Total	2,044		1,983

RQ1.1: *To what extent do the APIs in distributed systems and microservices suffer from poor design?*

Goal: To find whether APIs in distributed systems and microservices suffer from linguistic patterns and antipatterns.

RQ1.2: *What is the accuracy of the detection heuristics of linguistic patterns and antipatterns?*

Goal: To find the detection performance of the proposed detection heuristics.

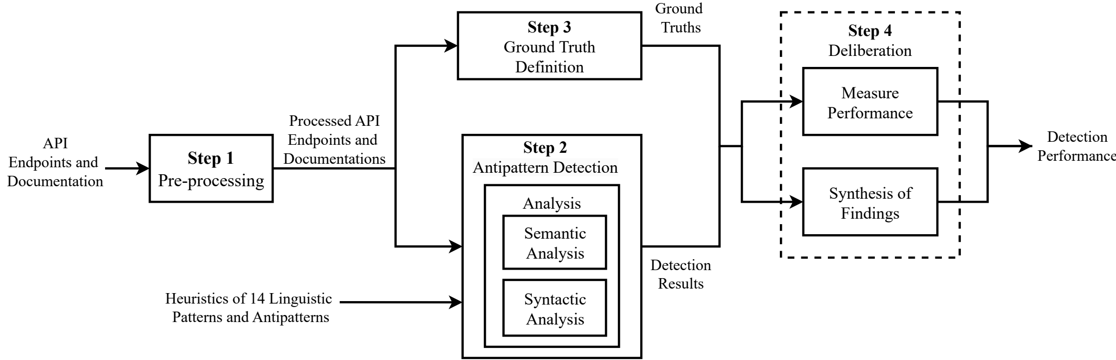

Figure 5.2: Overview of the linguistic patterns and antipatterns detection methods.

RQ1.3: *Which API category in distributed systems and microservices is more prone to poor linguistic design?*

Goal: To find the category of APIs more prone to poor linguistic design.

RQ1.4: *Which linguistic patterns and antipatterns are most common in APIs of distributed systems and microservices?*

Goal: To find the most commonly occurring linguistic patterns and antipatterns in APIs of distributed systems and microservices.

Detailed implementation of linguistic patterns and antipatterns detection are discussed in Chapter 6.

5.2.2 Impact Study

This Section describes the research design of the impact study. Jedlitschka et al. [28] proposed a software engineering experimental reporting structure, which we followed in our study. We also followed the reporting techniques proposed by Wyrich et al. [75] to report key characteristics of our experimental study. This impact study is inspired by the experiments conducted by Bogner et al. [10], where the authors assessed the impact of the REST design rule on the understandability of

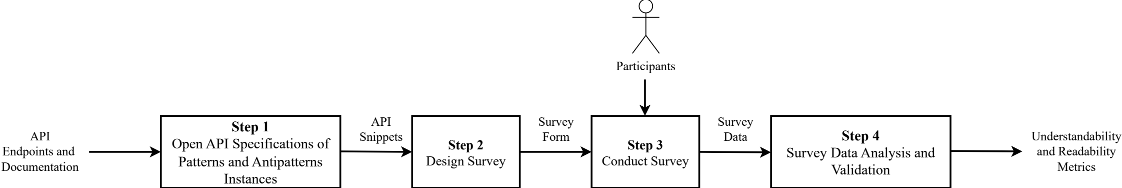

Figure 5.3: Overview of the impact study.

Table 5.2: Overview of impact study.

Aspect	Description
Goal	Study the effect of Web API design practices (linguistic design patterns and antipatterns) on the understandability and readability of Web APIs.
Study objects	10 linguistic design patterns (good design practices) and antipatterns (poor design practices) compiled from the literature, two functionally equivalent Web API snippets (one follows the linguistic design pattern, one follows the corresponding linguistic design antipatterns). Four newly proposed linguistic design patterns and antipatterns.
Participants	108 people with at least basic Web API design experience (participants include students and professionals from academia and industry).
Setting	Online survey via LimeSurvey[1].
Tasks	Answering comprehension questions (answer the purpose of a given Web API snippet) about API snippets for RQ2.1 (28 per participant), rating the difficulty in understandability of an API snippet for RQ2.2 (28 per participant), rating the difficulty in readability of a Web API snippets for RQ2.3 (28 per participant).
Dependent variables	Timed actual understandability (TAU) for RQ2.1, perceived difficulty rating in understandability for RQ2.4, perceived difficulty rating in readability for RQ2.3.
Other independent variables	Demographic attributes like Web API design and usage experience or current role (RQ2.4).
Design	Each participant was asked 28 comprehension questions (for both linguistic design patterns and antipatterns) in random order. Moreover, participants were also asked to rate the difficulty in understandability and readability of Web API snippets.

Web APIs. Figure 5.3 and Table 5.2 provide the overview of our research design. To investigate the impact of linguistic patterns and antipattern on the understandability and readability of APIs, we further divide RQ2 (as defined in Chapter 1) into four sub-research questions, as outlined below.

RQ2.1: *Which linguistic design patterns and antipatterns have an impact on the understandability of Web APIs?*

Goal: To find whether linguistic design patterns and antipatterns impact the understandability of APIs.

RQ2.2: *Do linguistic design patterns and antipatterns significantly influence software professionals' perceived difficulty in the understandability of Web APIs?*

Goal: To find whether linguistic patterns and antipatterns influence the perceived difficulty in the understandability of APIs.

RQ2.3: *Do linguistic design patterns and antipatterns significantly influence software professionals' perceived difficulty in the readability of Web APIs?*

Goal: To find whether linguistic patterns and antipatterns influence the perceived difficulty in the readability of APIs.

RQ2.4: *How do participant demographics influence the understandability and readability of Web APIs?*

Goal: To find whether the participants' demographic impacts the understandability and readability of APIs.

Detailed implementation of the impact study of linguistic patterns and antipatterns on understandability and readability of APIs in Chapter 6.

Chapter 6

Implementation

This chapter discusses both the detection method and the impact study. First, we discuss the implementation of the detection method in Section 6.1, followed by the overview of conducting the impact study in Section 6.2.

6.1 Detection of Linguistic Patterns and Antipatterns

This section briefly describes each step from Figure 5.2.

Step 1: Pre-processing

Figure 6.1 shows the overview of the pre-processing steps. We pre-process endpoint documentation and their parameters using standard NLP (Natural Language Processing) techniques before using them to define ground truth and detect linguistic patterns and antipatterns. In Step 1, we refine endpoint documentation by removing extra spaces, unknown characters, unknown symbols, and non-English charac-

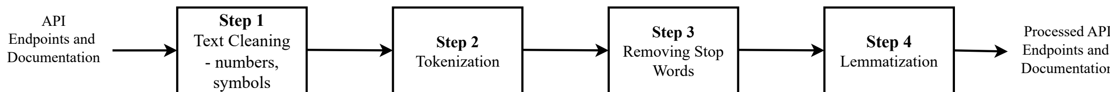

Figure 6.1: Pre-processing workflow (Step 1 from Figure 5.2).

ters. Then, we expand acronyms and decompose compound words to improve the readability and understandability of endpoint documentation and parameters. We gather a list of acronyms and compound words from the APIs and their corresponding split words for pre-processing purposes. In Step 2, we tokenize the text; in Step 3, we remove the stop words. Finally, in Step 4, we lemmatize and generate the processed API endpoints and documentation. Pre-processing, such as stop word removal, stemming, and lemmatization, are applied during the detection phase.

Step 2: Antipattern Detection

This step involves detecting 14 linguistic patterns and antipatterns described in Chapter 4. To detect these patterns and antipatterns, we follow similar detection methods employed in the literature [42, 45]. We also aim to improve several detection heuristics in terms of their detection performance, as discussed in Chapter 7.

Analysis of Linguistic Patterns and Antipatterns: We analyze the definition of patterns and antipatterns to explore their linguistic aspects. For example, detecting ✗*Non-pertinent Documentation* and ✗*Contextless Resource Names* antipattern requires semantic analysis of the endpoint and its documentation.

For example, the detection heuristic in Section 4.7 shows the heuristic employed to detect ✗*Non-pertinent Documentation* antipattern. We obtain domain-specific knowledge from the documentation of each endpoint (lines 2–3) to construct a topic model [67] in line 4. Subsequently, we extracted the nodes from the endpoint in line 5 and then computed the similarity between the nodes and the documentation

(line 6) using the LDA topic model. Then, we determine the average similarity value between each topic in the topic model and all nodes in an endpoint. An endpoint is assumed to have ✗*Non-pertinent Documentation* antipattern if the average similarity value is below the threshold (line 7). Conversely, ✓*Pertinent Documentation* pattern is reported if the similarity value equals or exceeds the threshold. We utilized a threshold of 0.5 for detection. The choice of 0.5 as the threshold aligns with the standard practice in cosine similarity [55] measurements.

The detection heuristic in Section 4.2 shows the detection heuristics for ✗*Contextless Resource Names* antipatterns. As described in Chapter 4, nodes in an endpoint should be semantically related [36]. To find the semantic relationship, we compare the nodes with the words in each topic of the LDA topic model based on the cosine similarity score. The LDA topic model from the documentation of endpoints is constructed in line 2. We extract the nodes from the endpoint in line 3. Suppose all the nodes of the endpoint fall into one topic of the LDA topic model. In that case, our detection algorithms identify that nodes are semantically related, i.e., ✓*Contextualized Resource Names*. In line 4, we calculate the similarity score of nodes against each topic of the LDA topic model. Suppose the average similarity score of nodes is below the threshold. In that case, all the nodes of the endpoints are not present in one topic of the topic model (line 5), i.e., ✗*Contextless Resource Names*. In contrast, if the average similarity score of nodes is above a certain threshold, then the detection algorithm identifies the endpoint as ✓*Contextualized Resource Names*, i.e., all nodes of the endpoint should fall into at least one topic of the topic model. Similar to ✗*Non-pertinent Documentation* antipattern heuristic, 0.5 is used as the threshold for identification.

Step 3: Ground Truth Definition

The definition of ground truth involves utilizing a random sampling approach to select 94 endpoints from a pool of 4,027 endpoints from 37 REST and 32 GraphQL APIs. This step aims to measure the performance of our implemented detection heuristics. The population comprises 56,378 endpoint instances (4,027 endpoints $\times$ 14 antipatterns). We chose a sample size of 1,316 queries (94 endpoints $\times$ 14 antipatterns) with a confidence interval of 10 and a confidence level of 95%.

Three professionals with expertise in REST and GraphQL API design were involved in the validation process. None of these professionals were part of the detection process, nor were the detection results shared and discussed with them to avoid bias. For the validation, we prepared online questionnaires using Google Forms[1], describing all ten patterns and antipatterns with appropriate examples to provide some background. We provided the HTTP method, endpoint, description, and parameters (if available) for each individual. We used majority voting to select whether an endpoint has a specific antipattern.

Step 4: Deliberation

Deliberation involves analyzing and synthesizing the findings. We also compare the detection results and the ground truth generated by experts to compute several performance metrics. To answer the defined research questions, we use the detection performance of the detection algorithm. To answer RQ2.1, we compile a detailed detection results table and use mosaic plots for REST and GraphQL APIs. For RQ2.2, we used various performance metrics such as accuracy, precision, recall, and F1-score. Finally, to answer RQ2.3 and RQ2.4, we visualize the results through the detection result table and stacked column chart, illustrating the proportion of

[1]https://forms.gle/E7h8RVRYg4umHTtU8

endpoints identified as antipatterns across the two API categories for distributed systems and microservices.

6.2 Impact Study

Figure 5.3 shows each step of our impact study. To examine the impact of linguistic patterns and antipatterns on API understandability and readability, we conduct a survey involving participants with API design experience and knowledge. We then analyze the survey data to address the defined research questions.

Step 1: OpenAPI Specification of Patterns and Antipatterns

To create Web API snippets from the API linguistic design patterns and antipatterns endpoint pairs, we utilized the OpenAPI[2] specification format. OpenAPI is one of the most popular ways to document Web APIs. API practitioners and researchers widely use and are familiar with OpenAPI specifications. We developed two OpenAPI documents, one with the endpoint examples following the linguistic design patterns and one with the endpoint examples following linguistic design antipatterns. We used the Swagger[3] editor to develop the OpenAPI documents. Each OpenAPI document contained 14 endpoints, one per linguistic design pattern or antipattern. We created screenshots of the graphical representation of each resource (endpoint and parameters), to present them to the participants with each task. Figure 6.2 shows the API snippet for ✓*Contextualized* and ✗*Contextless Resource Names* pattern and antipattern task.

[2]https://www.openapis.org/
[3]https://editor.swagger.io/

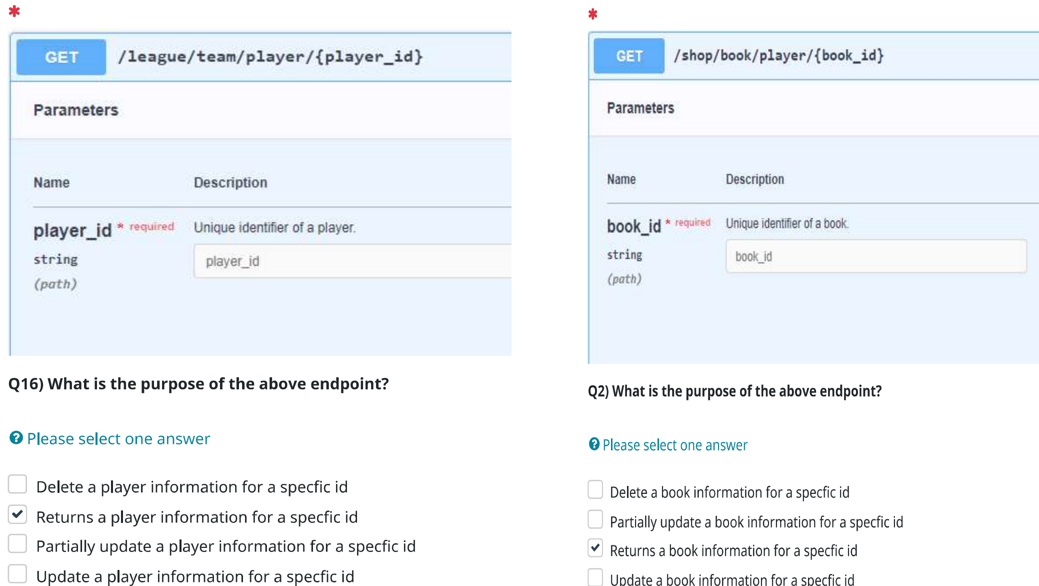

Figure 6.2: Example of a comprehension question based on ✓*Contextualized Resource Names* (left) and ✗*Contextless Resource Names* (right), with the correct answer marked.

Step 2: Design Survey

We conducted our survey via a web-based tool to reach a larger and more diverse audience. We used LimeSurvey[4] for our survey, which is an open-source survey tool. LimeSurvey provides many features that support several types of questions, are highly customizable, and allow measuring the duration per task or survey question. Duration per task is needed for our survey to calculate TAU (Timed Actual Understandability). Participants can use any computing device and web browser to access the survey hosted on LimeSurvey. A detailed description of the survey was also provided on the welcome page to guide the participants throughout the survey.

We used a mixed experimental design, combining elements of a between-subjects design [74] and a crossover design [70]. Such a mixed experimental design allowed us to take advantage of both experimental design techniques. Crossover design is a variant of a within-subjects design where every participant receives every linguistic

[4] https://www.limesurvey.org/

design pattern and antipattern at least once. However, the sequence in which they receive the patterns and antipatterns varies. Participants in our survey[5] worked on 14 tasks with the linguistic design patterns and 14 tasks with the linguistic design antipatterns. Allowing each participant to work on both patterns and antipatterns enhanced the robustness of our survey. To avoid familiarization effects or learning effects we randomized the order of the questions. Although randomized task ordering might lead to suboptimal task ordering of patterns and antipatterns (carryover or order effects), we aimed to gather the perspective of each participant on all 14 tasks [28]. This approach made our survey more concrete and avoided potential biases that could arise from creating multiple task groups. For instance, a participant might be highly familiar with one group of tasks but less so with another. Using a single-task group eliminated these limitations and reduced biases in our study. We had internal discussions and several iterations of testing with external reviewers before finalizing the survey design. Based on their feedback, we refined the tasks and survey language. We removed unnecessary questions to ensure participation time remained within $30 - 35$ minutes.

Survey Objects: In this survey, the objects under study were 14 linguistic design patterns and antipatterns in web API design. Chapter 4 provides definitions, examples, and detection heuristics of these linguistic design patterns and antipatterns. We extracted ten linguistic design patterns and antipatterns from several studies in the literature [43, 42, 45]. Additionally, we define four new linguistic design patterns and antipatterns. These four linguistic design patterns and antipatterns are defined from the existing web API design practices in the literature [36, 68]. All these linguistic design patterns and antipatterns have high importance and influence the maintainability of APIs [43, 42]. For each linguistic design pattern and antipattern, we also present a concrete API endpoint pair as an example. These API endpoints are from

[5] https://github.com/krishnodey/API-Quality/blob/master/Journal-Resources/Impact-Survey/Impact-Survey-Design.pdf

real-life APIs. For our survey, the selected real-world endpoints were modified to simplify them and ensure participants were not already familiar with the presented endpoints. During the dummy survey, we shared the linguistic design patterns and antipattern snippet pairs with external experts to validate them. Several snippets were adapted based on their feedback.

Tasks: The participants' main task was to examine and understand API snippets, which were presented using the graphical representation of the Swagger editor. To assess understandability and readability, participants answered one comprehension question per snippet. Each question followed a single-choice format with four randomly ordered options, only one of which was correct. For consistency, the answer choices remained the same for a given pattern or antipattern.

Using single-choice comprehension questions provided several advantages. While free-text responses could offer deeper insights, they are significantly more challenging to evaluate. Additionally, free-text responses require more effort from participants, potentially increasing dropout rates. They may also vary in detail, complicating analysis and affecting response times, which are a key part of our comprehension metrics. To maintain consistency, each snippet was paired with one of two predefined comprehension question types.

- **Return Value:** Participants were tasked to identify the endpoint's return value. More precisely, participants were asked to specify the entity type and whether it returned a single object or a collection.
- **Endpoint Purpose:** Participants were required to identify the purpose of the displayed endpoint, specifically the operation or functionality performed upon invocation.

After each comprehension question, participants were asked to rate the API snippet's

perceived difficulty of understandability and readability on a 5-point ordinal scale, from very easy (1) to very hard (5). The last snippet was also presented with rating questions so the participants would not have to rely on their memory.

Variables and Hypotheses: We used three dependent variables in our survey. Answering RQ2.1 requires a metric for understandability. To calculate this quality metric, we collected both the correctness and duration for each task per participant. Since each task had only one correct answer, correctness was represented as a binary variable, with 0 for incorrect and 1 for correct. The duration required for each task was recorded in seconds. To combine these two measures into a single variable, we used the TAU aggregation procedure [61, 10].

In our survey, TAU for participant p and task t was calculated as follows:

$$TAU_{p,t} = correctness_{p,t} \times \left(1 - \frac{duration_{p,t}}{max(duration_s)}\right) \tag{6.1}$$

TAU produces values between 0 and 1, where values closer to 1 indicate a higher degree of understandability. For incorrect answers, TAU is always 0. The task duration for the correct answer is normalized relative to the maximum duration recorded for that task. The normalized task duration is then inverted by subtracting it from one so that the faster the correct answer is found, the higher the TAU value. As a result, TAU provides aggregation of correctness and duration, accounting for differences between participants in the sample. Despite leading to unusual distributions (e.g., see Figures 7.10 and 7.11), we chose TAU as the dependent variable for RQ2.1. For RQ2.2, the dependent variable was the perceived difficulty in understandability, which was the rating participants provided after each task using a 5-point ordinal scale, ranging from very easy (1) to very hard (5). Similarly, for RQ2.3, the dependent variable was the perceived difficulty in readability.

Table 6.1: Null Hypotheses with Their Alternatives for the Four Confirmatory RQs.

RQs	**Metric**	**Null Hypothesis**	**Alternative Hypothesis**
RQ2.1	Timed actual understandability (TAU)	H_0^1: API snippets following linguistic design patterns and antipatterns are equally or less understandable.	H_1^1: API snippets following linguistic design patterns are more understandable than snippets following antipatterns.
RQ2.2	Perceived Difficulty in Understandability of APIs	H_0^2: API snippets following linguistic design antipatterns are rated equally or more difficult to understand than snippets following patterns.	H_1^2: API snippets following linguistic design antipatterns are rated as more difficult to understand than snippets following patterns.
RQ2.3	Perceived Difficulty in Readability of APIs	H_0^3: API snippets following linguistic design antipatterns are rated equally or more difficult to read than snippets following patterns.	H_1^3: API snippets following linguistic design antipatterns are rated as more difficult to read than snippets following patterns.
RQ2.4	Relationships with Demographic Attributes	H_0^4: Software professionals with API design experience find snippets following linguistic design patterns equally difficult and easier to understand and read than snippets following antipatterns.	H_1^4: Software professionals with API design experience find snippets following linguistic design patterns easier to understand and read than snippets following antipatterns.

The controlled independent variable was the linguistic design patterns and antipatterns in the respective snippet. Additionally, we collected uncontrollable independent variables for RQ2.4, such as participants' current roles, years of experience with APIs, and knowledge of the Richardson maturity model.

Based on these variables, we formulated hypotheses for the confirmatory questions RQ2.1 (difference in actual understandability between the linguistic design pattern and antipattern), RQ2.2 (difference in perceived understandability), RQ2.3 (difference in perceived readability), and RQ2.4 (relationships with demographic attributes), as shown in Table 6.1. In each case, we hypothesized that snippets following the linguistic design patterns would result in significantly better outcomes than those following linguistic design antipatterns. Each 14 linguistic design pattern and antipattern was tested individually to identify their impact on understandability and readability.

Step 3: Conduct the Survey

Participation in the LimeSurvey online survey was open for approximately three months (23rd October 2024 - 31st January 2025). The survey URL was widely advertised in our network at the beginning, and during that time, the survey was available online. A welcome page with some basic study information was shown to experiment participants at the beginning of the survey. We explained that the only prerequisite for participation was a basic understanding of web API design (e.g., REST), and the purpose of the survey was to investigate the understandability and readability of web APIs. We stated that the survey should ideally be completed in a single sitting and not take more than $30 - 35$ minutes. As an added incentive, we promised to donate $1 to charity for each participant.

We also thoroughly described the task and mentioned that participants would be presented with 28 snippets. For each snippet, they will be asked to answer a multiple-choice question. After each comprehension question, they would be asked to rate how difficult it was to understand and read the API snippet. Lastly, they would also be asked to answer a few demographic questions.

Finally, we also mentioned the privacy policy and that the survey is anonymous. Survey data would only be used for research purposes. Participation was voluntary, and participants could exit the survey at any time. They were required to provide consent to these terms before starting the survey.

Once participants accepted the terms, they were forwarded to the tasks. For each of the 28 snippets, participants had to answer one comprehension question and two rating questions. The first rating question was how difficult it was to understand the API snippet, and the second one was how difficult it was to read the endpoint. API snippet was also available for the rating questions. Each participant had to answer 54 questions, one comprehension question, and two rating questions for each of the

28 snippets.

Finally, the participant answered demographic questions, such as their country of origin, current role, technical API perspective (API user /client developer, API developer /designer, or both), years of professional experience with APIs, and knowledge of the Richardson maturity model. Additionally, we offered a free-text space for participants to provide any feedback or comments about the survey. We also requested participants to share the survey URL with their colleagues.

Step 4: Survey Data Analysis

We first export all the responses as a CSV file to analyze the recorded response data. Then, we perform data cleaning and transformation steps presented in the following:

- Removing incomplete responses.
- Addressing and standardizing free-text responses.
- Transform text to numerical values (1 = correct answer, 0 = incorrect answer).
- Generate new columns for demographic analysis and assign binary values. New columns include *is_Student* and *is_Academia* (an academic professional).

Cleaning and transformation of the data did not result in any substantial ambiguity. However, since TAU is sensitive to outliers in task duration, we analyzed the duration of all comprehension questions. The response was excluded from the calculation if the duration was less than five seconds or exceeded four minutes. Responses under five seconds or over four minutes indicated improper reading or lack of full concentration. If participants answered before five seconds or took more than four minutes, they were likely not fully concentrated on the task. After cleaning and transforming

Table 6.2: Effect Size Interpretation Based on Cohen's d.

Effect Size (d)	Interpretation
$d < 0.2$	Very Small Effect
$0.2 \leq d < 0.5$	Small Effect
$0.5 \leq d < 0.8$	Medium Effect
$0.8 \leq d < 1.2$	Large Effect
$1.2 \leq d < 2.0$	Very Large Effect
$d \geq 2.0$	Huge Effect

data, we import the clean data in an R[6] script. Using the R script, we perform basic data transformation, calculate TAU, and provide general descriptive analysis. From the descriptive analysis, we generate different diagrams for visualization. We also perform the hypothesis testing for all four research questions. Firstly, for RQ2.1, RQ2.2, and RQ2.3, we use the Shapiro-Wilk test [65] to visualize the data distribution. For all the dependent variables, data was not distributed normally, i.e., p-value $\ll 0.05$. As a result, we performed a non-parametric Wilcoxon-Mann-Whitney test [39]. We used the Holm-Bonferroni adjustment to address the multiple comparison problems, which in our instance was the testing of 14 linguistic design patterns and antipatterns [63]. We rejected the null hypothesis and accepted the alternative when the adjusted p-value fell below our targeted significance level of 0.05. We also computed Cohen's d to assess the effect size of accepted hypotheses [13]. Effect sizes of Cohen's d according to Sawilowsky [60] are presented in Table 6.2.

To test the hypothesis of RQ2.4, we used a correlation matrix. The correlation matrix provides a visual representation of the relationship among demographic information. We use Kendall's Tau [30] test with the correlation of the identified pairs. Kendall's Tau is more robust and permissive when making assumptions about the data. Finally, in order to investigate the possibility of predictive modeling and to further examine the combined effects of demographic characteristics, we also employed linear regression[7].

[6] https://www.r-project.org/

[7] https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/lm

Chapter 7

Results

This chapter presents and discusses the findings of this thesis. We use these findings to answer the research questions (RQs) defined in Chapter 1.

7.1 RQ1: Prevalence of Linguistic Patterns and Antipatterns in APIs

This section presents our detection results and answers our sub-research questions (RQ1.x) for RQ1 defined in Section 5.2. Figures 7.1 and 7.2 depict the mosaic plots of the detection results for 14 linguistic patterns and antipatterns on 69 APIs. The number of endpoints analyzed for an API corresponds to the height of the boxes (row-wise). The ratio of endpoints classified as patterns and antipatterns is displayed by the width of the white and black boxes (column-wise) for each antipattern. As shown in Figures 7.1 and 7.2, the most prevalent antipattern in APIs is ✗*Unversioned Endpoint*, while other common antipatterns are ✗*Amorphous Endpoint*, ✗*Pluralized Nodes*, and ✗*Inconsistent Resource Archetype*.

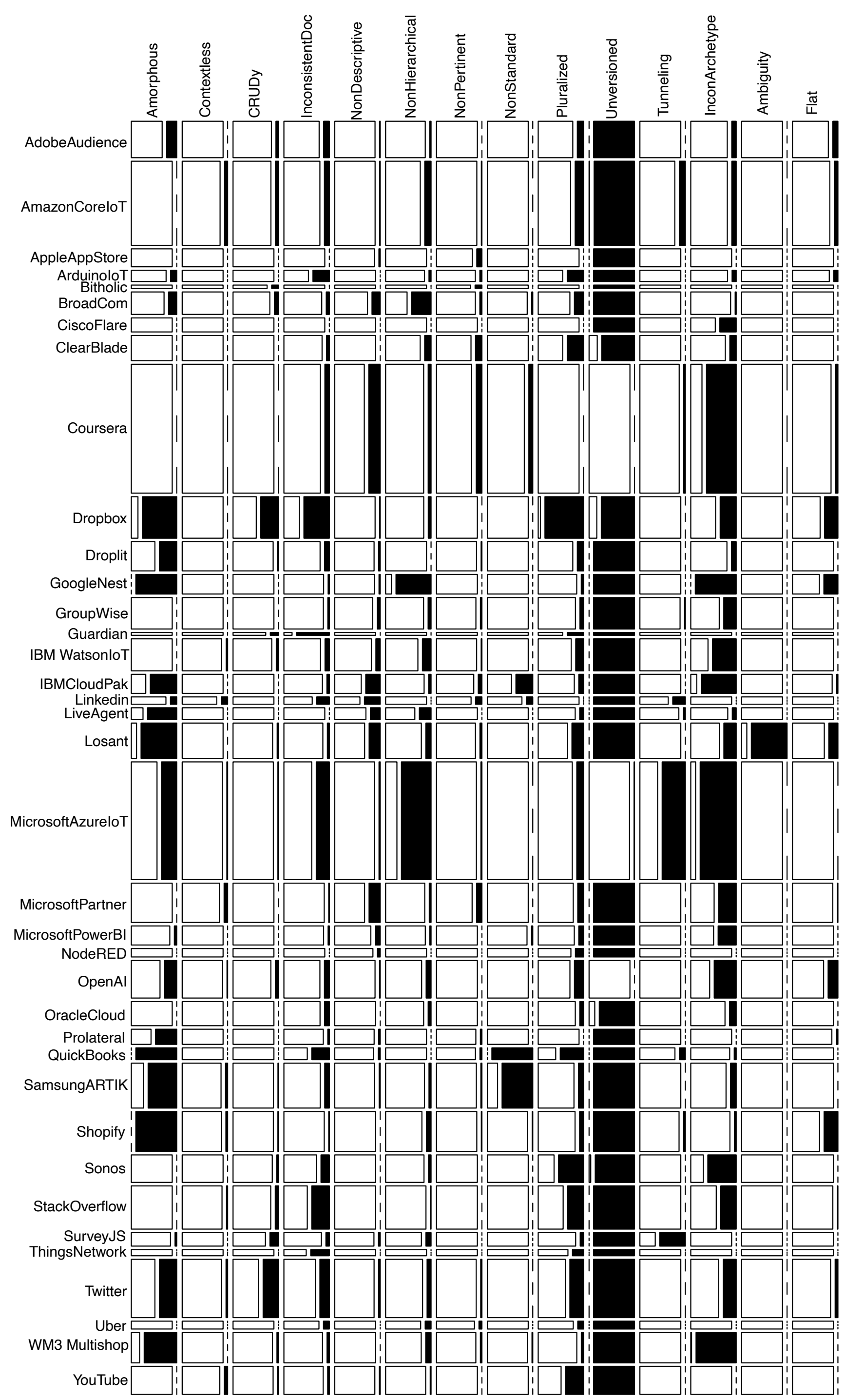

Figure 7.1: Detection of Patterns and Antipatterns in REST APIs. The black portion represents antipatterns and the white portion represents patterns.

In REST APIs, Dropbox, Microsoft Azure IoT, Uber, and X (formerly Twitter) have a significant presence of linguistic antipattern. Dropbox and Microsoft Azure IoT

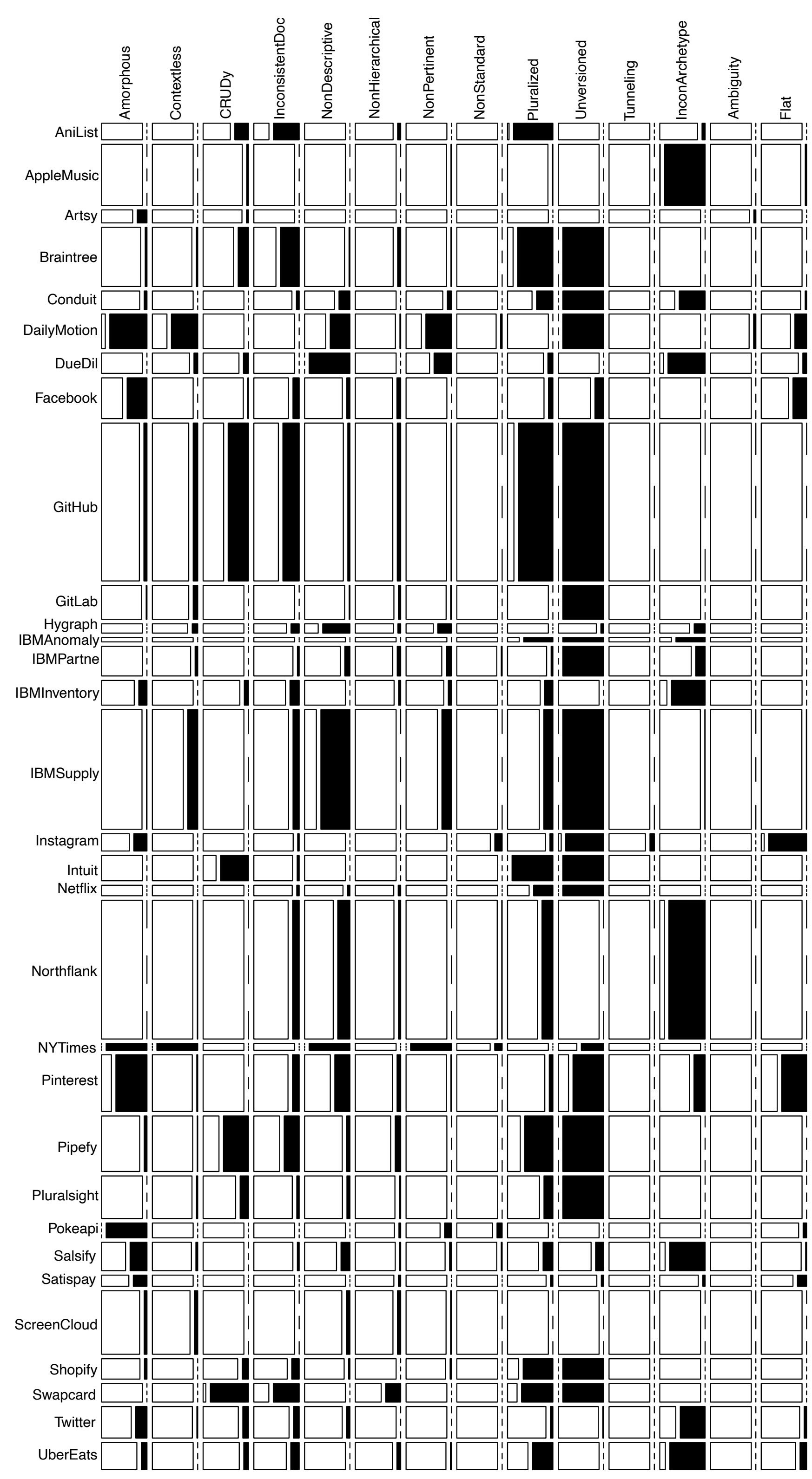

Figure 7.2: Detection of Patterns and Antipatterns in GraphQL APIs. The black portion represents antipatterns and the white portion represents patterns.

APIs are prone to ✗*Unversioned Endpoint*, ✗*Pluralized Nodes*, ✗*Amorphous Endpoint* and ✗*Non-standard* linguistic antipatterns. The presence of these linguistic antipatterns suggests inconsistencies in API endpoint design and the use of non-standard characters in endpoint naming. APIs such as LiveAgent, Shopify, GoogleNest, and Coursera show a moderate presence of linguistic antipatterns. These APIs suffer from specific linguistic antipatterns. For instance, Shopify API suffer from ✗*CRUDy Endpoint*, ✗*Unversioned Endpoint* antipatterns. On the other hand, APIs such as Open AI, IBM Watson IoT, Oracle Cloud, and Node RED follow linguistic patterns with minor exceptions. For example, Open AI API follows eight linguistic patterns while violating six linguistic patterns. Similarly, IBM Watson IoT adheres to six linguistic patterns and violates eight linguistic patterns. However, the number of violations among those eight linguistic patterns is relatively small. In contrast to the Open AI and IBM Watson IoT APIs, Oracle Cloud and Node-RED adhere to more linguistic patterns. Oracle Cloud follows ten linguistic patterns and violates four linguistic antipatterns, while Node-RED adheres to 12 linguistic patterns and violates two linguistic antipatterns. APIs like Dropbox, Microsoft Azure IoT, and X should follow good design principles (linguistic patterns) and design more user-friendly APIs for their clients. Open AI and Oracle Cloud APIs set good examples by adhering to good API design principles (linguistic patterns).

Among the GraphQL APIs, Facebook, IBM Supply Chain, and Pinterest exhibit a significant presence of linguistic antipatterns. ✗*Inconsistent Documentation*, ✗*Non-standard Endpoint*, ✗*Unversioned Endpoint*, and ✗*CRUDy Endpoint* antipatterns are prevalent in these APIs. For example, Facebook and IBM Supply Chain seem to suffer from ✗*Inconsistent Documentation*, ✗*Unversioned Endpoint*, ✗*Contextless Resource Names*, and ✗*Amorphous Endpoint* antipatterns. Braintree, DailyMotion, Pipefy, GitHub, and Shopify APIs also suffer from linguistic antipatterns. For example, Braintree and GitHub API noticeably suffer from ✗*Unversioned End-*

point, ✗*Pluralized Nodes*, ✗*CRUDy Endpoint* and ✗*Inconsistent Documentation* antipatterns. DailyMotion suffers from ✗*Amorphous Endpoint*, ✗*Contextless Resource Names*, ✗*Non-descriptive Endpoint*, ✗*Unversioned Endpoint*, and ✗*Flat Endpoint* antipatterns. Pipefy and Shopify APIs suffer from ✗*CRUDy Endpoint*, ✗*Inconsistent Documentation*, ✗*Pluralized Nodes*, and ✗*Unversioned Endpoint* linguistic antipatterns. In contrast, Apple Music, Artsy, GitLab, and IBM Anomaly Detection largely adhere to linguistic design patterns, prioritizing consistent documentation and versioning. Apple Music and GitLab follow 12 linguistic patterns while violating two. Meanwhile, Artsy and the IBM Anomaly Detection API adhere to 11 linguistic patterns and violate three. Proper use of versioning and standard endpoint issues remain prevalent across many GraphQL APIs, which could potentially lead to long-term maintainability challenges. Moreover, some APIs lack consistent and precise documentation, hindering developer experience and integration.

The detailed detection results for 14 patterns and antipatterns are shown in Tables 7.1 and 7.2. Each column shows the detection instances for all 14 pairs of patterns and antipatterns, and each row shows the number of endpoints detected as patterns and antipatterns for the APIs. Our results suggest that 74% of the endpoints contain ✗*Unversioned Endpoint* antipatterns. In contrast, 98% of the endpoints follow ✓*Contextualized Resource Names* pattern. In GraphQL APIs, 58% of the endpoints follow ✗*Unversioned Endpoint* antipatterns, and 99% follow ✓*Standard Endpoint* design. Moreover, almost 100% endpoints follow ✓*Parameter Tunneling* and ✓*Identifier Annotation* patterns.

Table 7.1: Detection Results on 4,027 Endpoints from 37 REST APIs for 14 Patterns and Antipatterns.

API Name	✗Amorphous	✓Tidy	✗Contextless	✓Contextual	✗CRUDy	✓Verbless	✗Inconsistent	✓Consistent	✗Non-Descriptive	✓Descriptive	✗Non-Hierarchical	✓Hierarchical	✗Non-Pertinent	✓Pertinent	✗Non-Standard	✓Standard	✗Pluralized	✓Singularized	✗Unversioned	✓Versioned	✗Parameter-Tunneling	✓Parameter-Adherence	✗Inconsistent-Archetype	✓Consistent-Archetype	✗Identifier-Ambiguity	✓Identifier-Annotation	✗Flat	✓Structured
Adobe Audience	16	49	0	65	4	61	9	56	1	64	2	63	0	65	0	65	10	55	65	0	0	65	7	58	0	65	8	57
Amazon AWS	0	150	11	139	11	139	18	132	2	148	23	127	2	148	0	150	31	119	148	2	22	128	16	134	0	150	13	137
Apple App Store	0	32	0	32	0	32	0	32	1	31	0	32	4	28	0	32	0	32	32	0	0	32	0	32	0	32	0	32
Arduino IoT Cloud	3	17	0	20	0	20	8	12	0	20	1	19	1	19	0	20	8	12	20	0	0	20	2	18	0	20	2	18
Bitholic	0	6	0	6	1	5	0	6	0	6	0	6	1	5	0	6	0	6	6	0	0	6	0	6	0	6	0	6
BroadCom	8	32	0	40	4	36	3	37	8	32	19	21	2	38	1	39	9	31	40	0	0	40	1	39	0	40	0	40
Cisco Flare	0	25	0	25	0	25	0	25	0	25	0	25	0	25	0	25	0	25	25	0	0	25	10	15	0	25	0	25
ClearBlade	0	45	0	45	0	45	3	42	0	45	7	38	7	38	0	45	18	27	36	9	0	45	7	38	0	45	0	45
Coursera	0	230	0	230	0	230	25	205	64	166	16	214	32	198	22	208	0	230	0	230	9	221	167	63	0	230	13	217
Dropbox	62	12	0	74	32	42	46	28	1	73	5	69	0	74	0	74	70	4	60	14	0	74	29	45	0	74	24	50
Droplit	22	30	0	52	1	51	6	46	2	50	0	52	1	51	0	52	8	44	52	0	0	52	6	46	0	52	0	52
Google Nest	35	0	0	35	0	35	1	34	1	34	30	5	0	35	0	35	2	33	35	0	0	35	35	0	0	35	12	23
GroupWise	0	56	0	56	0	56	2	54	4	52	4	52	2	54	0	56	5	51	56	0	1	55	17	39	0	56	0	56
Guardian	0	5	0	5	1	4	4	1	0	5	0	5	0	5	0	5	2	3	5	0	0	5	0	5	0	5	0	5
IBM Watson IoT	0	57	2	55	3	54	6	51	6	51	12	45	1	56	0	57	11	46	57	0	0	57	33	24	0	57	0	57
IBM Cloud Pak	22	12	0	34	0	34	2	32	12	22	2	32	3	31	14	20	4	30	34	0	0	34	29	5	0	34	1	33
Linkedin	2	11	2	11	0	13	4	9	5	8	0	13	2	11	2	11	0	13	13	0	4	9	0	13	0	13	0	13
LiveAgent	15	6	0	21	0	21	0	21	5	16	6	15	0	21	0	21	2	19	21	0	1	20	2	19	0	21	0	21
Losant	55	8	0	63	2	61	3	60	17	46	8	55	3	60	0	63	18	45	63	0	0	63	19	44	55	8	14	49
Microsoft Azure IoT	78	132	4	206	3	207	67	143	6	204	152	58	5	205	0	210	35	175	2	208	119	91	185	25	0	210	0	210
Microsoft Partner	0	70	6	64	0	70	1	69	19	51	3	67	9	61	0	70	10	60	70	0	0	70	30	40	0	70	1	69
Microsoft Power BI	2	32	0	34	0	34	1	33	4	30	1	33	0	34	1	33	4	30	34	0	0	34	15	19	0	34	0	34
Node	0	15	0	15	0	15	0	15	1	14	0	15	0	15	0	15	3	12	15	0	0	15	0	15	0	15	0	15
OpenAI	20	47	1	66	5	62	8	59	1	66	8	59	0	67	0	67	15	52	0	67	0	67	36	31	0	67	16	51
OracleCloud	0	43	0	43	0	43	5	38	0	43	3	40	0	43	0	43	4	39	37	6	0	43	7	36	0	43	0	43
Prolateral Core	14	13	0	27	0	27	1	26	1	26	0	27	1	26	0	27	0	27	27	0	0	27	0	27	0	27	1	26
QuickBooks	21	0	0	21	0	21	9	12	1	20	0	21	1	20	21	0	12	9	21	0	3	18	1	20	0	21	0	21
Samsung ARTIK	56	24	4	76	1	79	9	71	2	78	4	76	2	78	60	20	10	70	80	0	0	80	11	69	0	80	0	80
Shopify	71	0	3	68	1	70	1	70	0	71	8	63	1	70	1	70	7	64	71	0	3	68	3	68	0	71	24	47
Sonos	0	49	0	49	2	47	10	39	0	49	3	46	0	49	0	49	30	19	47	2	0	49	34	15	0	49	0	49
StackOverflow	0	77	2	75	6	71	33	44	0	77	1	76	0	77	0	77	30	47	77	0	0	77	29	48	0	77	1	76
SurveyJS	1	23	0	24	5	19	2	22	1	23	3	21	0	24	0	24	2	22	24	0	15	9	0	24	0	24	0	24
Things Network	0	11	0	11	0	11	5	6	0	11	0	11	0	11	0	11	3	8	11	0	0	11	0	11	0	11	0	11
X (formerly Twitter)	44	60	4	100	39	65	24	80	4	100	10	94	3	101	0	104	35	69	104	0	0	104	33	71	0	104	7	97
Uber	0	14	0	14	0	14	2	12	0	14	2	12	1	13	0	14	2	12	14	0	0	14	0	14	0	14	0	14
WM3 Multishop	43	11	0	54	2	52	3	51	0	54	7	47	2	52	1	53	3	51	54	0	0	54	53	1	0	54	0	54
YouTube	0	50	4	46	1	49	2	48	1	49	1	49	0	50	0	50	22	28	50	0	0	50	0	50	0	50	0	50
Total	**590**	**1,454**	**43**	**2,001**	**124**	**1,920**	**323**	**1,721**	**170**	**1,874**	**341**	**1,703**	**86**	**1,958**	**123**	**1,921**	**425**	**1,619**	**1,506**	**538**	**177**	**1,867**	**817**	**1,227**	**55**	**1,989**	**137**	**1,907**
Percentage (%)	**29%**	**71%**	**2%**	**98%**	**6%**	**94%**	**16%**	**84%**	**8%**	**92%**	**17%**	**83%**	**4%**	**96%**	**6%**	**94%**	**21%**	**79%**	**74%**	**26%**	**9%**	**91%**	**40%**	**60%**	**3%**	**97%**	**6%**	**93%**

Table 7.2: Detection Results on 4,027 Endpoints from 32 GraphQL APIs for 14 Patterns and Antipatterns.

API Name	✗*Amorphous*	✓*Tidy*	✗*Contextless*	✓*Contextual*	✗*CRUDy*	✓*Verbless*	✗*Inconsistent*	✓*Consistent*	✗*Non-Descriptive*	✓*Descriptive*	✗*Non-Hierarchical*	✓*Hierarchical*	✗*Non-Pertinent*	✓*Pertinent*	✗*Non-Standard*	✓*Standard*	✗*Pluralized*	✓*Singularized*	✗*Unversioned*	✓*Versioned*	✗*Parameter-Tunneling*	✓*Parameter-Adherence*	✗*Inconsistent-Archetype*	✓*Consistent-Archetype*	✗*Identifier-Ambiguity*	✓*Identifier-Annotation*	✗*Flat*	✓*Structured*
AniList	0	27	0	27	9	18	17	10	0	27	2	25	0	27	0	27	26	1	0	27	0	27	2	25	0	27	0	27
AppleMusic	0	99	0	99	4	95	0	99	0	99	0	99	1	98	0	99	1	98	0	99	0	99	98	1	0	99	3	96
Artsy	5	16	0	21	1	20	0	21	0	21	0	21	0	21	0	21	0	21	0	21	1	20	0	21	0	21	0	21
Braintree	4	92	3	93	24	72	44	52	2	94	7	89	1	95	0	96	83	13	96	0	0	96	0	96	0	96	0	96
Conduit	2	28	0	30	0	30	2	28	8	22	0	30	3	27	0	30	12	18	30	0	0	30	19	11	0	30	1	29
DailyMotion	51	5	36	20	0	56	0	56	27	29	1	55	35	21	2	54	0	56	56	0	0	56	0	56	3	53	16	40
DueDil	0	33	3	30	4	29	0	33	33	0	0	33	14	19	0	33	4	29	0	33	0	33	30	3	0	33	3	30
Facebook	32	34	0	66	1	65	10	56	5	61	6	60	0	66	0	66	7	59	14	52	0	66	0	66	0	66	22	44
GitHub	19	237	28	228	127	129	102	154	13	243	21	235	2	254	0	256	216	40	256	0	0	256	0	256	0	256	0	256
GitLab	1	54	6	49	0	55	0	55	1	54	3	52	1	54	0	55	0	55	55	0	0	55	0	55	0	55	0	55
Hygraph	0	15	2	13	0	15	3	12	10	5	1	14	5	10	0	15	0	15	1	14	0	15	4	11	0	15	0	15
IBM Anomaly	0	7	0	7	0	7	0	7	0	7	0	7	5	2	7	0	0	7	5	2	0	7	0	7	0	7	0	7
IBM Partner	0	48	3	45	0	48	2	46	6	42	4	44	6	42	1	47	2	46	48	0	0	48	11	37	0	48	0	48
IBM Inventory	8	32	0	40	4	36	9	31	0	40	2	38	3	37	0	40	8	32	0	40	0	40	33	7	0	40	0	40
IBM Supply Chain	2	192	47	147	0	194	29	165	138	56	0	194	45	149	0	194	43	151	194	0	0	194	1	193	0	194	1	193
Instagram	9	19	0	28	0	28	1	27	0	28	0	28	0	28	5	23	2	26	26	2	3	25	0	28	0	28	26	2
Intuit	0	41	0	41	28	13	2	39	0	41	0	41	0	41	0	41	41	0	41	0	0	41	0	41	0	41	0	41
Netflix	0	17	0	17	0	17	1	16	1	16	1	16	0	17	0	17	8	9	17	0	0	17	0	17	0	17	0	17
Northflank	0	225	2	223	0	225	36	189	67	158	12	213	0	225	2	223	60	165	0	225	0	225	200	25	0	225	0	225
NY Times	11	0	11	0	0	11	0	11	11	0	0	11	11	0	2	9	0	11	6	5	0	11	0	11	0	11	0	11
Pinterest	70	22	2	90	0	92	15	77	34	58	8	84	1	91	0	92	8	84	69	23	0	92	25	67	0	92	56	36
Pipefy	6	84	1	89	55	35	33	57	7	83	12	78	0	90	0	90	62	28	90	0	0	90	0	90	0	90	0	90
Pluralsight	0	69	2	67	14	55	4	65	5	64	2	67	0	69	0	69	15	54	69	0	0	69	0	69	0	69	0	69
Pokeapi	24	0	0	24	0	24	0	24	0	24	1	23	4	20	3	21	0	24	0	24	0	24	0	24	0	24	2	22
Salsify	19	27	2	44	0	46	2	44	10	36	1	45	2	44	1	45	11	35	9	37	0	46	40	6	0	46	1	45
Satispay	6	12	0	18	0	18	0	18	0	18	1	17	0	18	0	18	1	17	1	17	0	18	1	17	0	18	4	14
ScreenCloud	7	96	8	95	0	103	0	103	9	94	8	95	2	101	0	103	0	103	0	103	0	103	0	103	0	103	0	103
Shopify	2	31	0	33	5	28	6	27	1	32	0	33	1	32	0	33	24	9	33	0	0	33	0	33	0	33	0	33
Swapcard	0	30	0	30	28	2	19	11	0	30	11	19	0	30	0	30	23	7	30	0	0	30	0	30	0	30	0	30
X (formerly Twitter)	14	37	1	50	7	44	7	44	4	47	0	51	0	51	0	51	3	48	1	50	0	51	31	20	0	51	3	48
UberEats	6	38	0	44	5	39	9	35	0	44	4	40	1	43	0	44	22	22	0	44	0	44	38	6	0	44	7	37
Yelp Fusion	6	12	0	18	3	15	1	17	2	16	0	18	1	17	2	16	0	18	0	18	1	17	0	18	1	17	0	18
Total	**304**	**1,679**	**157**	**1,826**	**319**	**1,664**	**354**	**1,629**	**395**	**1,588**	**109**	**1,874**	**139**	**1,844**	**16**	**1,967**	**688**	**1,295**	**1,151**	**832**	**3**	**1,980**	**544**	**1,439**	**4**	**1,979**	**148**	**1,835**
Percentage (%)	**15%**	**85%**	**8%**	**92%**	**16%**	**84%**	**18%**	**82%**	**20%**	**80%**	**5%**	**95%**	**7%**	**93%**	**1%**	**99%**	**35%**	**65%**	**58%**	**42%**	**0%**	**100%**	**27%**	**73%**	**0%**	**100%**	**7%**	93%

RQ1.1: Poor Linguistic Design in APIs

RQ1.1 investigates the presence of poor linguistic design quality in APIs. This thesis only considers REST and GraphQL APIs of distributed systems and microservices. From Tables 7.1 and 7.2, the most prevalent antipattern in REST APIs is ✗*Unversioned Endpoint*, constituting 74% of the detected antipatterns. Moreover, REST APIs generally follow good API design principles for linguistic patterns. For example, 98% follow ✓*Contextualized Resource Names* patterns, 97% follow ✓*Identifier Annotation* patterns, 94% of the endpoints follow ✓*Standard Endpoint* and ✓*Verbless Endpoint* patterns, and 99% of the endpoints have ✓*Structured Endpoint*. This suggests that REST API developers tend to follow good API design practices.

In GraphQL APIs, the most prevalent antipatterns include ✗*Unversioned Endpoint* (58% of the endpoints), ✗*Pluralized Nodes* (35%) and ✗*Inconsistent Resource Archetype Names* (35%). For other linguistic patterns, GraphQL follows design practices similar to those of REST APIs. For example, GraphQL APIs commonly follow linguistic patterns like ✓*Standard Endpoint* (99% of the endpoints), ✓*Parameter Adherence* (99.84%), ✓*Identifier Annotation* (99.79%), ✓*Contextualized Resource Names* (92%), ✓*Hierarchical Nodes*(95%), and ✓*Pertinent Documentation*(93%). Thus, GraphQL API developers also follow good API design practices, except for ✗*Unversioned Endpoint*, ✗*Pluralized Nodes*, and ✗*Inconsistent Resource Archetype Names* antipatterns.

From Figure 7.1, in REST APIs, the second most common antipattern is ✗*Inconsistent Resource Archetype Names* and the third most common antipattern is ✗*Amorphous Endpoint*. Furthermore, ✗*Pluralized Nodes* and ✗*Inconsistent Documentation* antipatterns are also prevalent in REST. Among the other REST APIs, the majority of APIs also follow the ✓*Verbless Endpoint* pattern, except for Dropbox, SurveyJS, and X (formerly Twitter). Similarly, most APIs adopt the ✓*Identifier Annnotation*

pattern, excluding the Losant APIs. ✓*Standard Endpoint* pattern is common in all APIs, except for Coursera, IBM Cloud Pak, Linkedin, QuickBooks, and SamsungARTIK.

For GraphQL APIs, as shown in Figure 7.2, nearly all APIs exhibit ✗*Unversioned Endpoint*, except for Apple Music, Artsy, DailyMotion, IBM Inventory, Northflank, ScreenCloud, and UberEats. Similarly, ✗*Pluralized Node* and ✗*Inconsistent Resource Archetype Names* antipattern are prevalent in almost all the APIs. Moreover, we also found that the presence of ✗*Inconsistent Documentation* is widespread across most GraphQL APIs, except for Apple Music, Artsy, Daily Motion, DueDil, GitLab, PokéAPI, Screen Cloud and Satispay. Furthermore, developers often use CRUDy term for resource naming, i.e., a large portion of the GraphQL APIs demonstrate ✗*CRUDy Endpoint* antipattern.

RQ1.1 Summary: Poor linguistic designs (i.e., linguistic antipatterns) are present in the APIs of distributed systems and microservices. Thus, despite the wide adoption of these APIs, they still lack quality design.

RQ1.2: Accuracy of Detection Algorithms

RQ1.2 aims to investigate the detection accuracy of our detection algorithms. The detection accuracy for each of the 14 patterns and antipatterns is shown in Table 7.3. On a set of 94 endpoints (i.e., 94×10 instances of antipattern), our detection algorithms achieved an average detection accuracy of 94.07%, precision of 82.77%, and recall of 87.35% (thus, average F1-score of 88.06%). Validation results suggest that our detection algorithms outperform state-of-the-art detection methods [42, 45, 15].

The performance metrics can be influenced by how developers, in this case, three professionals, understand and interpret a word based on their experience and knowl-

edge. From Table 7.3, we can observe that ✗*Contextless Resource Names*, ✗*Non-descriptive Endpoint*, ✗*Non-hierarchical Nodes*, ✗*Pluralized Nodes* and ✗*Inconsistent Resource Archetype Names* have significantly low detection performance compared to other linguistic antipatterns. For example, 12 endpoints were detected as instances of ✗*Non-hierarchical Nodes* antipattern, but only six were validated as ✗*Non-hierarchical Nodes*. For instance, the endpoint `/v19.0/{application-id}/button_auto_detection_device_selection` from Facebook API was detected as ✗*Non-hierarchical Nodes* antipattern, however, based on the majority voting, the manual validation (ground truth definition) identified the endpoint as ✓*Hierarchical Nodes* pattern. Consequently, ✗*Non-hierarchical Nodes* antipattern exhibits a low precision, recall, and F1-score due to such interpretation conflicts among the professionals. Similarly, ✗*Pluralized Nodes* antipattern has low precision, recall and F1-score. Detection tool detected 27 endpoint as ✗*Pluralized Nodes*, however only 14 were validated as ✗*Pluralized Nodes*.

In contrast, ✗*Amorphous Endpoint*, ✗*Unversioned Endpoint*, ✗*Parameter Tunneling* antipattern have excellent accuracy, precision, recall, and F1 score. Such good detection could be due to the easier interpretation of these antipatterns. These antipatterns are easy for professionals to validate and detect. For example, endpoint `/destinations/limits` from Adobe Audience Manager API does not have any versioning information, while endpoint `/v2/lists/:id/tweets` from X (formerly Twitter) API has versioning information, which is very easy to understand and detect. Similarly, the endpoint `/devices/smoke_co_alarms/device_id/name` from Google Nest API contains an underscore, which results in the ✗*Amorphous Endpoint* antipattern. The presence of underscores makes it apparent for professionals and tools to detect and understand.

In summary, complex linguistic antipatterns such as ✗*Non-hierarchical Nodes* and ✗*Pluralized Nodes* exhibited poor performance due to varying interpretations among

Table 7.3: Performance of the detection algorithms. P: Positive, N: Negative, Pre: Precision, Rec: Recall, F1: F1 Score.

Antipatterns	P	N	TP	FP	TN	FN	Accuracy	Pre	Rec	F1
✗Amorphous Endpoint	34	60	31	3	60	0	96.81%	91.18%	100%	93.91%
✗Contextless Resource Names	3	91	2	1	85	6	92.55%	66.67%	25%	77.51%
✗CRUDy Endpoint	18	76	16	2	76	0	97.87%	88.89%	100%	93.16%
✗Inconsistent Documentation	14	80	11	3	69	11	85.11%	78.57%	50%	81.71%
✗Non-pertinent Documentation	4	90	3	1	90	0	98.94%	75%	100%	85.32%
✗Non-descriptive Endpoint	3	91	2	1	89	2	96.81%	66.67%	50%	78.96%
✗Non-hierarchical Nodes	12	82	6	6	80	2	91.49%	50%	75%	64.66%
✗Non-standard Endpoint	8	86	7	1	83	3	95.74%	87.50%	70%	91.44%
✗Pluralized Nodes	27	67	14	13	61	6	79.79%	51.85%	70%	62.86%
✗Unversioned Endpoint	73	21	73	0	21	0	100%	100%	100%	100%
✗Parameter Tunneling	1	93	1	0	93	0	100%	100%	100%	100%
✗Inconsistent Archetype	24	70	12	12	69	1	86.17%	50%	92.31%	63.28%
✗Identifier Ambiguity	3	91	3	0	90	1	98.94%	100%	75%	99.47%
✗Flat Endpoint	43	51	40	3	51	0	96.81%	93.02%	100%	94.88%
Average							94.07%	82.77%	87.35%	88.06%

professionals and sometimes the detection method's inability to identify them accurately.

RQ1.2 Summary: Our detection algorithms achieved an average accuracy of 94.07%, precision of 82.77%, recall of 87.35%, and F1-score of 88.06%. Compared to the state-of-the-art methods, our detection algorithms yield better detection performance.

RQ1.3: API Category Prone to Poor Linguistic Design

RQ1.3 aims to identify the API category that is more susceptible to poor linguistic design. Figure 7.3 shows the proportion of endpoints detected as antipatterns in REST and GraphQL APIs. The figure suggests that linguistic antipatterns are prevalent in both REST and GraphQL APIs. The figure also suggests that the proportion of antipattern instances in a specific category varies for each of the 14 antipatterns. Figure 7.3 shows that REST APIs contain more antipatterns than GraphQL APIs. More specifically, endpoints from GraphQL APIs had 4,331 instances of antipatterns (out of 27,762 = 1,983 endpoints $\times$ 14 antipatterns). In comparison, endpoints

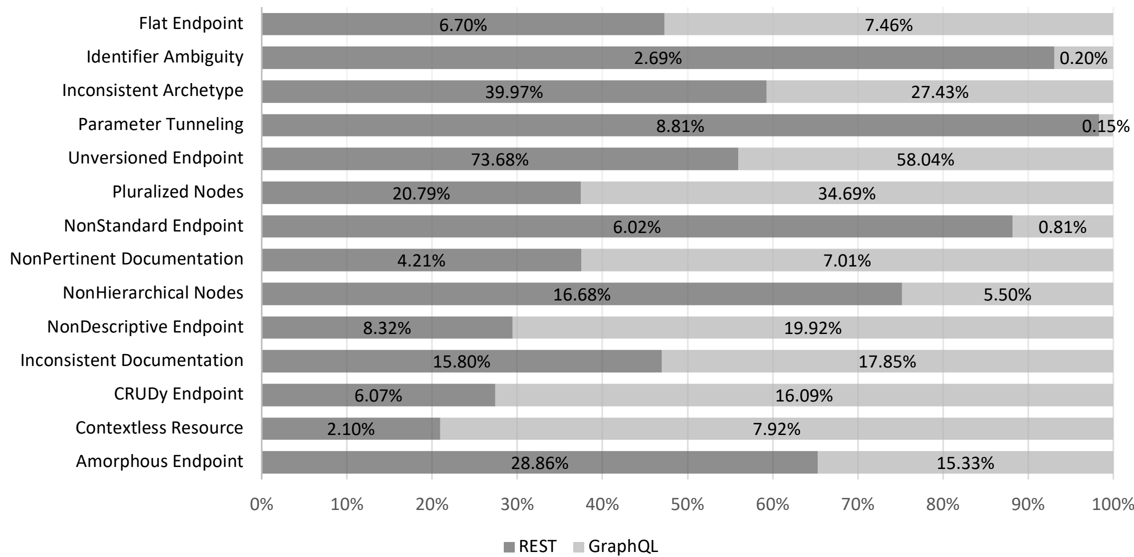

Figure 7.3: Prevalence of antipatterns in REST and GraphQL APIs.

from REST APIs had 4,917 instances of antipatterns (out of 28,616 instances = 2,044 endpoints × 14 antipatterns), which suggests that overall 17% of the REST endpoints contain antipatterns, in contrast to 15.6% for GraphQL endpoints. Thus, REST APIs have a slightly higher proportion of antipattern instances than GraphQL APIs.

RQ1.3 Summary: According to our detection results, antipatterns are more prevalent in REST compared to GraphQL APIs, i.e., GraphQL APIs are well designed compared to REST APIs in terms of linguistic quality, although the margin of difference is very small.

RQ1.4: Most Common Linguistic Patterns and Antipatterns

We aim to find which linguistic patterns and antipatterns are more common in APIs of distributed systems and microservices through RQ1.4. From Table 7.1, Table 7.2 and Figure 7.3, the most common antipatterns are ✗*Unversioned Endpoint*, ✗*Amorphous Endpoint*, ✗*Pluralized Endpoint* and ✗*Inconsistent Resource Archetype*

Names. In contrast, common good design practices among the APIs are ✓*Descriptive Endpoint*, ✓*Contextualized Resource Names*, ✓*Pertinent Documentation*, ✓*Parameter Adherence* and ✓*Identifier Annotation*. Linguistic antipatterns that are rare in REST APIs are ✗*Contextless Resource Names* (2% of endpoints), ✗*Non-pertinent Documentation* (4% of endpoints), and ✗*Identifier Annotation* (3% of endpoints). Furthermore, only 6% of endpoints follow ✗*CRUDy Endpoint*, ✗*Non-standard Endpoint*, and ✗*Flat Endpoint* antipatterns. On the other hand, for GraphQL APIs, ✗*Contextless Resource Names* (8% of endpoints), ✗*Non-standard Endpoints* (1% of endpoints), ✗*Pertinent Documentation* (7% of endpoints), ✗*Parameter Tunneling* (0.14% of endpoints) and ✗*Identifier Annotation* (0.21% of endpoints) antipatterns are rare. Moreover, in REST and GraphQL APIs, ✓*Standard Endpoints* and ✓*Hierarchical Nodes* are common good practices.

RQ1.4 Summary: Most commonly occurring antipatterns are ✗*Unversioned Endpoint*, ✗*Amorphous Endpoint*, ✗*Pluralized Nodes*, and ✗*Inconsistent Resource Archetype Names*, i.e., developers are not concerned with versioning of the endpoints. Moreover, using uppercase, underscores, file extensions, and trailing slashes are common in endpoint design, which are poor design choices. Furthermore, the use of plural nouns and inconsistent resource archetypes for endpoint naming is a common practice in API design.

7.1.1 Discussions on Detection

Our detection results suggest that APIs of distributed systems and microservices are not always well-designed, i.e., have antipatterns. In this section, we discuss the detection of some of the common antipatterns.

In total, 58 out of 69 analyzed APIs lack version information. Versioning is critical to API design as it helps API and client developers manage their endpoints as they

evolve. Also, with a continuous evolution of endpoints, ✗*Unversioned Endpoints* might break clients and become difficult to manage. For example, an endpoint from Dropbox API `/files/create_folder_batch/check` does not use versioning, thus our detection algorithm and experts during ground truth generation identified it as ✗*Unversioned Endpoint* antipattern. In contrast, an endpoint `/v1/me/ratings/songs/{id}` from Apple Music API uses versioning, hence identified as ✓Versioned Endpoint pattern.

When nodes of an endpoint are not semantically related, it is considered as ✗*Contextless Resource Names* antipattern. For example, in the X API (formally known as Twitter), the endpoint `/v2/spaces/search` was detected as ✗*Contextless Resource Names* by our detection algorithm because there is a lack of semantic relationship between nodes *spaces* and *search*. In contrast, an endpoint `/api/user/reg` from Clear Blade API was detected as ✓*Contextualized Resource Names* because the detection algorithm finds semantic relationship among *api, user, reg*. The use of semantically related terms while designing endpoints improves readability and understandability. On the other hand, semantically dissimilar terms in the endpoint could be misleading and hard to understand.

✗*Inconsistent Documentation* antipatterns occur when the documentation of the endpoint is in contradiction with the HTTP method, i.e., an appropriate HTTP method was not implemented. In the Broad Com API, the endpoint `/api/getAccess` uses POST HTTP method with documentation *'Returns the Authentication Token X-AccessToken, as part of response headers, if the provided user name and password is correct'*, which is conflicting because HTTP GET method should be used to retrieve resources and HTTP POST method should be used to create resources. Thus, our detection algorithm identified this endpoint as an ✗*Inconsistent Documentation* antipattern. Similar issues can be observed in one endpoint `/bulk/devices/remove` from IBM Watson IoT API where the HTTP method POST is in contradiction with

the documentation *'Delete multiple devices. Delete multiple devices, each request can contain a maximum of 512 kB'*. The presence of an ✗*Inconsistent Documentation* antipattern significantly reduces the understandability, as the endpoint does not do what its documentation says. In contrast, a consistent endpoint is easy to understand and is not misleading.

✗*Amorphous Endpoint* antipattern is one the most prevalent antipatterns, with 45 out of 69 APIs containing this antipattern. An endpoint `/devices/cameras/device_id/snapshot_url` from Google Nest API was detected as an instance of ✗*Amorphous Endpoint* as our detection algorithm found underscores in the endpoint. An endpoint should not use uppercase letters, underscores, file extensions, or trailing slashes, which may limit the understandability of endpoints. In contrast, an endpoint `/datasources/bulk-delete` of Adobe Audience Manager API demonstrates a ✓*Tidy Endpoint*. Tidy endpoints are very concise and help the client developers understand their purpose.

✗*Flat Endpoint* antipattern occurs when (/) is not used to break down nodes in a hierarchical manner. An endpoint `/createTableRecordInRestrictedTable` from Pipefy API demonstrates ✗*Flat Endpoint* antipattern because resources are not broken down with (/), which makes it difficult for client developers to comprehend and use the endpoint. Conversely, an endpoint `/users/<userID>/properties` from Samsung ARTIK Cloud API is broken down to smaller resource names, which demonstrates ✓*Structured Endpoint* pattern. The use of (/) to break down resources into smaller names helps API developers and clients understand the purpose of an endpoint.

✗*Identifier Ambiguity* antipattern occurs when an endpoint does not use curly braces or colons to indicate an identifier. The use of curly braces or colons makes it easy for client developers to find identifiers in an endpoint. An endpoint `/dashboards/DA`

SHBOARD_ID from Losant API does not use any symbol to differentiate the identifier from the rest of the nodes. In cases where the URI is long, it may be confusing for client developers to identify the relevant identifier. In contrast, endpoint /codeadmin/failed/{systemKey} from ClearBlade API uses curly braces to indicate the identifier.

Antipatterns are prevalent in APIs of distributed systems and microservices, as discussed above. This thesis presents interesting findings that could help API providers and client developers identify and improve their API endpoint design.

Comparison of the Detection Performance:

Table 7.4 compares the detection performance with current state-of-the-art detection methods. Several studies have been performed for Cloud services [48, 52, 50, 12, 11, 58], which are mostly based on syntactic analysis. We studied 69 APIs (REST and GraphQL), where we performed both syntactic and semantic analysis of more than 4,027 endpoints. We performed the detection of 14 linguistic antipatterns and their corresponding linguistic patterns to assess the linguistic quality of APIs.

A comparison can be made based on the type of API analyzed, the number of APIs examined, the number of endpoints considered, the number of antipatterns analyzed, and the performance of the detection method/algorithms. Our detection method achieved an average F1-score of 88.06%. The state-of-the-art approaches SARA [42] achieved an average F1-score of 80.9% and DOLAR [43] achieved 79.5%. SARAv2 [45] (an extension of the SARA approach) achieved an average F1-score of 64% on linguistic antipattern detection on a similar sample validation size (i.e., 94). The detection algorithms implemented in this thesis achieved a better performance than state-of-the-art methods (DOLAR, SARA, SARAv2, RESTRuler) regarding both F1-score and accuracy. The detection algorithms in this thesis out-

performed DOLAR and SARA in terms of F1-score by a margin of 8.6% and 7.2%. Our detection algorithms also outperformed SARAv2 by a margin 24.1% in terms of the F1-score. Such improvement in detection performance could be due to several reasons, including the use of *Cosine Similarity* instead of *Second-Order Similarity* [33] metric to capture the similarity between words, improving the overall semantic analysis.

The study conducted by Brabra et al. achieved an average F1-score of 99% [12]. Other studies also show an average F1-score between 79.5% and 99%. However, these studies are (i) either focused on different types of APIs (such as Cloud services), (ii) have a limited number of analyzed APIs (ranging from 2 to 20 APIs), (iii) detect a relatively lower number of linguistic patterns and antipatterns (between 2 and 28 antipatterns), or (iv) they perform only fine-grained syntactic analyses (e.g., those for OCCI patterns and antipatterns) [52, 12]. Moreover, all other studies are focused on REST APIs, and we also detect linguistic patterns and antipatterns of GraphQL APIs. Considering the highly semantic nature of our automatic analysis using our detection method and the subjective validation of results by experts, we recognize that the interpretation of patterns and antipatterns may vary. This variation is due to the full degree of freedom in deciding patterns and antipatterns. Therefore, we consider an average F1-score of more than 88.06% to be acceptable in the domain of natural language processing [64].

7.1.2 Implications for Developers

Application developers usually review the API documentation provided by the API developers before they consume the API. API design quality is essential to facilitate their use by application developers. Application developers would opt for well-designed APIs compared to poorly designed APIs because of the ease of understand-

Table 7.4: Comparison with state-of-the-art detection methods.

Study	**Year**	**API Types**	**APIs Analyzed**	**Endpoints Analyzed**	**Antipatterns Rules**	**F1-score**
Parrish [48]	2010	REST, Graph API	2	-	-	n/a
Wei et al. [73]	2015	REST	-	-	-	n/a
Palma et al. (DOLAR) [43]	2015	REST	15	350	5	79.5%
Brabra et al. [12]	2016	REST	6	-	28	99%
Petrillo et al. [52]	2016	REST	3	-	73	n/a%
Rodriguez et al. [58]	2016	REST	-	-	-	n/a%
Haupt et al. [24]	2017	REST	-	-	-	n/a%
Palma et al. (SARA) [42]	2017	REST	18	555	6	80.94%
Petrillo et al. (CLOUDLEX) [50]	2018	REST	16	23,000	2	71.03%
Palma et al. (SARAv2) [45]	2022	REST IoT	19	1,102	9	64%
Bogner et al. (RESTRuler) [9]	2024	REST	-	2,300	14	n/a
Dey et al. [15]	2024	REST, GraphQL	33	1,655	10	85.98%
This study	2025	REST, GraphQL	69	4,027	14	88.06%

ing and use. This thesis aims to find empirical evidence of linguistic antipatterns in APIs of distributed systems and microservices. Our findings suggest that linguistic antipatterns exist in both categories of APIs (i.e., REST and GraphQL), which will help API developers address the existing linguistic antipatterns and help them improve the overall design quality of their APIs. Well-designed APIs will attract more consumers and improve the overall user experience. Results from this thesis suggest that one of the most commonly occurring antipatterns is ✗*Unversioned Endpoint*, which means API developers are not providing versioning of endpoints, which may make API evolution and maintenance difficult. Inconsistent use of singular and plural nouns (i.e., ✗*Pluralized Nodes*) while designing another common practice in real-world APIs. Moreover, improper use of singular/plural nouns and verbs is also common while indicating Document, Collection, Store, and Controller in API endpoints ✗*Inconsistent Resource Archetype Names*. Our other observations include: ✗*Amorphous Endpoint*, ✗*Non-standard Endpoint*, and ✗*CRUDy Endpoint*, i.e., API developers commonly include amorphous design, non-standard characters and CRUDy verbs in endpoint design. In summary, API developers could use our findings to improve the overall design quality of their APIs.

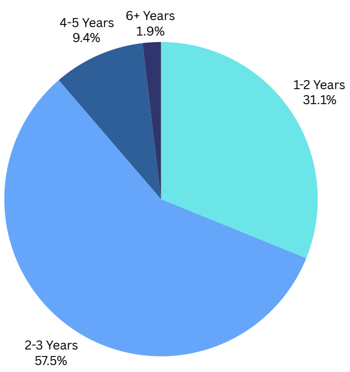

Figure 7.4: Distribution of the experience of the participants.

7.2 RQ2: Impact the Understandability and Readability

In RQ2, we aim to assess the impact of linguistic patterns and antipatterns on the understandability and readability of APIs. This section first presents general statistics about participants. Then, it provides the results for each sub-research question (RQ2.x), starting with descriptive statistics, followed by hypothesis testing or correlation results.

Participant Demographics

After performing the cleaning procedure mentioned in Section 6.2, we had 108 complete responses. We cleaned out 168 incomplete responses from the total number of 276 responses. Figure 7.4 shows the distribution of participants based on their years of experience in API design. Overall, participants with complete and valid responses had between one to nine years of experience with API Design (e.g., REST), with a median experience of two years. Figure 7.6 shows the distribution of the profession of the participants. Among 108 participants, 44 were Software Engineers/Developers, 41 were Students, 15 were Professors/Teachers, seven were Researchers, and one was Software Tester. In summary, 45 participants were from industry, and 63 were from

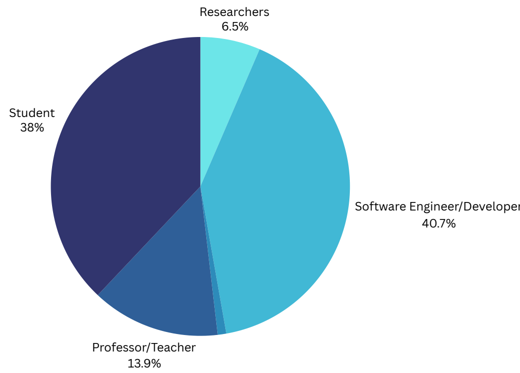

Figure 7.5: Distribution of the profession of the participants.

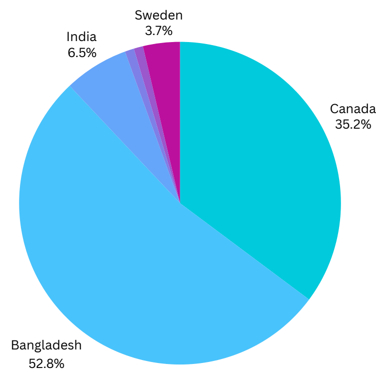

Figure 7.6: Distribution of the country of the participants.

academia.

Figure 7.6 shows the distribution of participants by country. The participants were from Bangladesh (57), Canada (38), India (7), Sweden (4), Ireland (1), and Pakistan (1). Among 45 industry personnel, 44 were software developers, and one was software tester. Two of the industry personnel were API testers, and others were API users (14), both API users and client developers (58), both API users and developers (26), and both API developers and designers (7). All 44 students were API users and client developers.

Although most participants are familiar with API usage and development, only 40 had knowledge of the Richardson Maturity Model (See Appendix A). The median maturity rating is 2; among these 40 participants, most of them selected level 2 and level 3. A few participants were level 1, and no one selected level 0. Such statistics

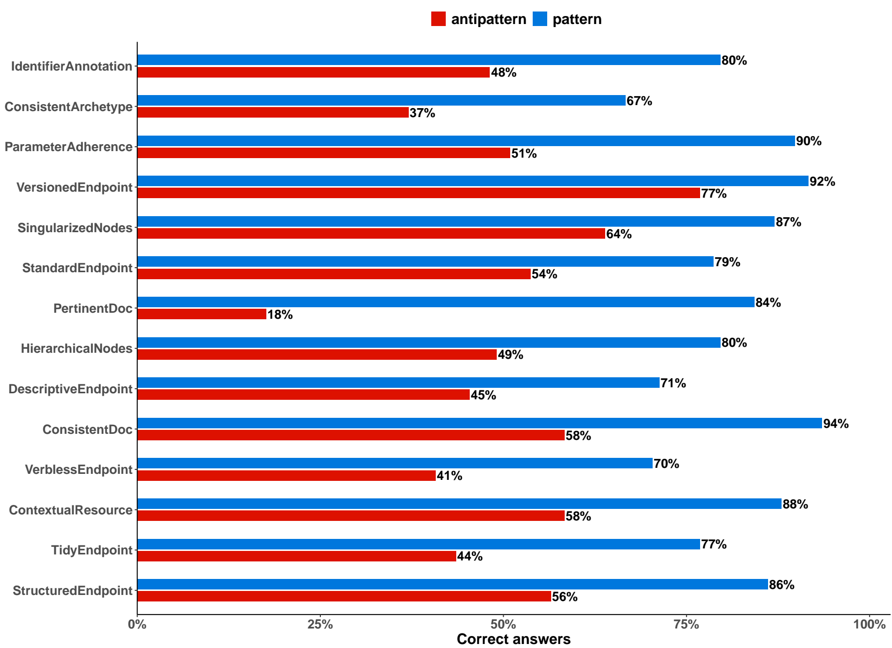

Figure 7.7: Comparison of correctness between the linguistic design patterns and antipatterns (higher is better). The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns.

may indicate that the Richardson maturity model may not be widely known, and HATEOAS is not considered an important requirement for designing Web APIs by professionals [34].

RQ2.1: Impact on Understandability

In RQ2.1, we aim to determine which linguistic design patterns and antipatterns significantly impact the understandability of the Web APIs. We measured the understandability of Web APIs from the comprehension task of the survey and TAU values. Figures 7.7, 7.8, and 7.9 respectively present the results for the percentage of correct answers, the mean time required to answer the comprehension task, and the TAU values for both patterns and antipatterns. Table B.1 and B.2 in the Appendix provide a detailed descriptive statistics.

For all 28 comprehension tasks, the participants performed significantly better on

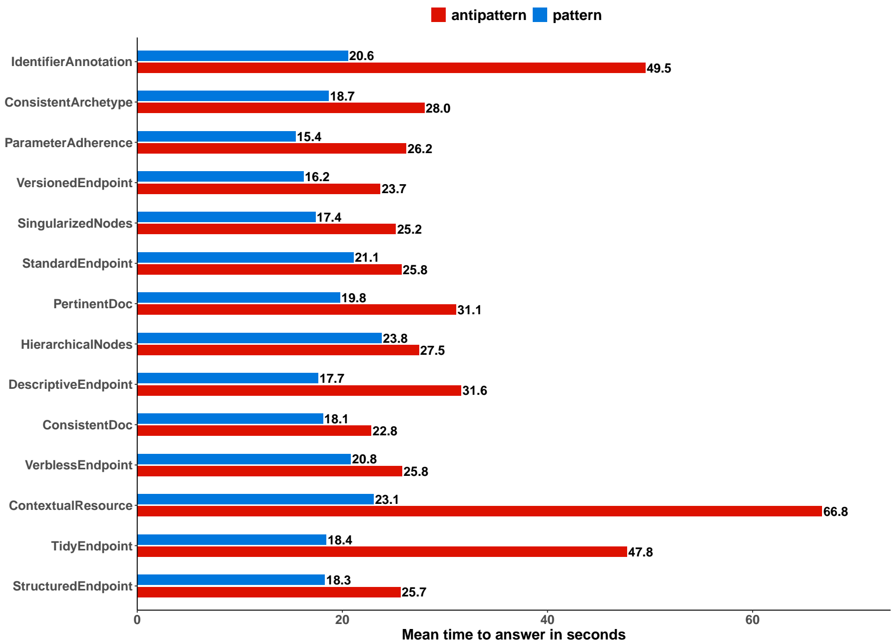

Figure 7.8: Comparison of duration between the linguistic design patterns and antipatterns (lower is better). The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns.

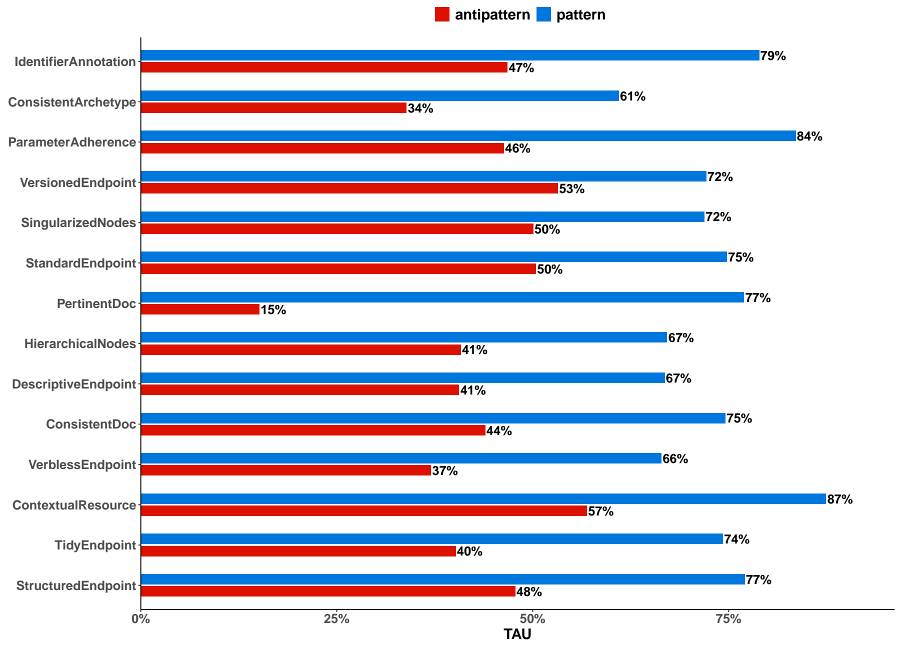

Figure 7.9: Comparison of TAU between the linguistic design patterns and antipatterns (higher is better). The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns.

tasks following linguistic design patterns than tasks following linguistic design antipattern. Figure 7.7 also shows a significant difference between tasks with linguistic design patterns and antipatterns in terms of correctness. Tasks following linguistic design patterns such as ✓*Pertinent Documentation* (84% for pattern vs. 18% for antipattern), ✓*Parameter Adherence* (90% for pattern vs. 51% for antipattern), and ✓*Consistent Documentation* (94% for pattern vs. 58% for antipattern) have significant differences in terms of correctness. Figure 7.8 shows that tasks following linguistic antipatterns required more time than tasks following linguistic patterns for all 14 pairs of linguistic patterns and antipatterns. Tasks following ✓*Identifier Annotation* (20.6s vs 49.5s), ✓*Contextual Resource Names* (23.1s vs 66.8s), ✓*Tidy Endpoint* (18.4s vs 25.7s) patterns required substantially less time than their corresponding antipatterns. In other cases, linguistic design patterns and antipatterns pairs ✓*Hierarchical Nodes vs. Non-Hierarchical Nodes* (23.8s for pattern vs. 27.5s for antipattern), ✓*Standard Endpoint vs. Non-Standard Endpoint* (21.1s for pattern vs. 25.8s for antipattern) and ✓*Consistent Documentation vs. Inconsistent Documentation* (18.1s for pattern vs. 22.8s for antipattern) have a similar response time.

Figure 7.9 shows the significant difference between the mean TAU for linguistic design patterns and antipatterns. This implies that tasks following linguistic design patterns are easier to understand, and tasks following linguistic design antipatterns are more difficult to understand. ✓*Pertinent Documentation* (77% for pattern vs. 15% for antipattern), ✓*Parameter Adherence* (84% for pattern vs. 46% for antipattern), and ✓*Contextualized Resource Names* (87% for pattern vs. 57% for antipattern) have significant differences of TAU values. The other 11 linguistic patterns and antipattern pairs also have noticeable differences in TAU values. Such difference suggests that tasks following linguistic patterns are easier to understand than those following antipatterns.

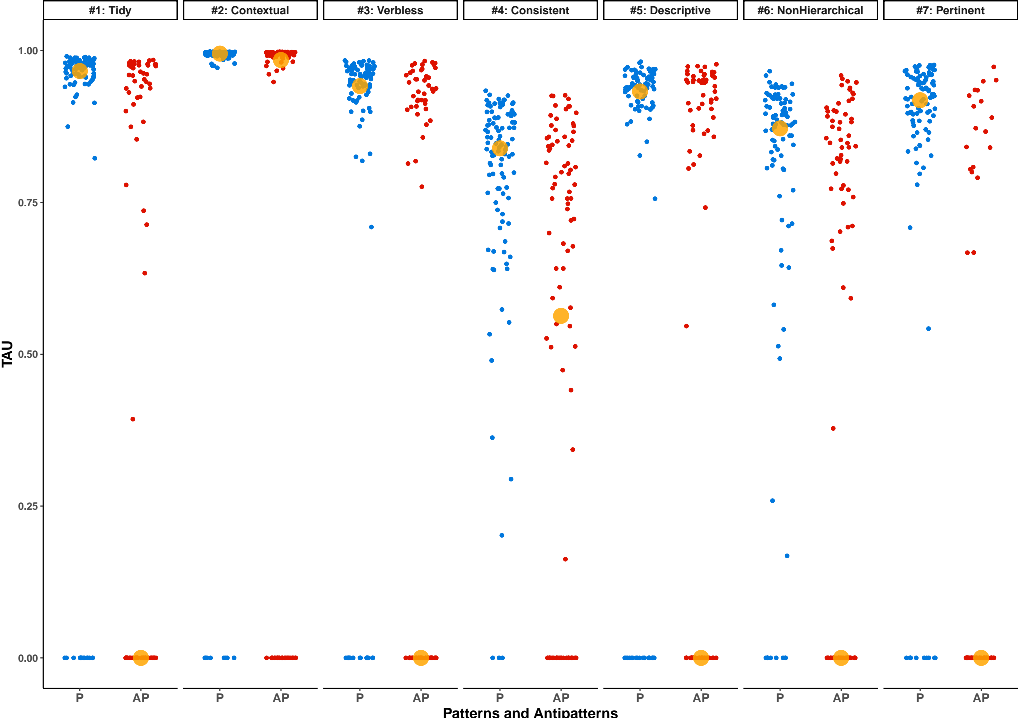

Figure 7.10: TAU distributions for ✓*Tidy Endpoint*, ✓*Contextualized Resource Names*, ✓*Verbless Endpoint*, ✓*Consistent Documentation*, ✓*Descriptive Endpoint*, ✓*Non-Hierarchical Nodes*, and ✓*Pertinent Documentation* along with their respective linguistic design antipatterns. P: Linguistic design pattern, AP: Linguistic design antipattern.

To further illustrate the experimental performance, we generated strip plots of TAU distributions for all linguistic patterns and antipatterns. These plots facilitate the identification of unusual distributions and enable the comparisons between linguistic patterns and antipatterns. Figures 7.10 and 7.11 show the distribution of TAU for all 14 linguistic design patterns and their corresponding antipatterns. The dots at the bottom represent the incorrect answers ($TAU = 0$), and the yellow circle shows the median values of TAU. All 14 tasks following linguistic design patterns have higher mean TAU distribution than tasks following linguistic antipatterns.

In Figure 7.10, linguistic design patterns and antipatterns pairs ✓*Tidy Endpoint vs. Amorphous Endpoint*, ✓*Verbless Endpoint* vs. ✗*CRUDy Endpoint*, ✓*Descriptive Endpoint* vs. ✗*Non-descriptive Endpoint*, and ✓*Pertinent Documentation* vs.

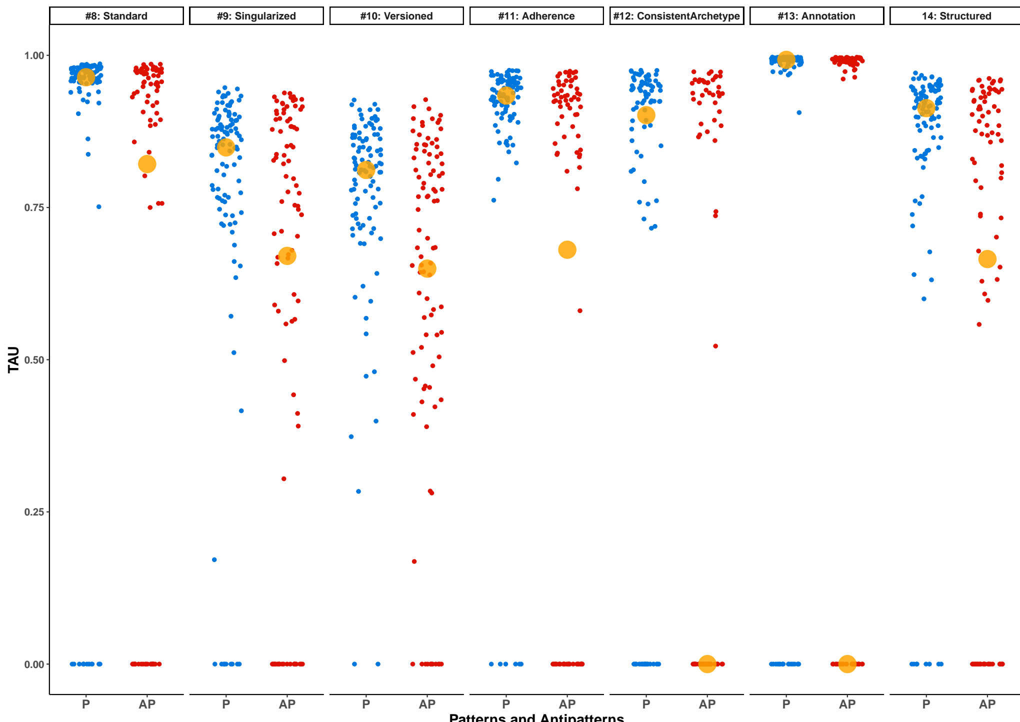

Figure 7.11: TAU distributions for ✓*Standard Endpoint*, ✓*Singularized Nodes*, ✓*Versioned Endpoint*, ✓*Parameter Adherence*, ✓*Consistent Resource Archetype Names*, ✓*Parameter Annotation* and *Structured Endpoint* along with their respective linguistic design antipatterns. P: Linguistic design pattern, AP: Linguistic design antipattern.

✗*Non-pertinent Documentation* exhibit significant differences in average TAU distributions. Similarly in Figure 7.11, linguistic patterns and antipatterns pairs ✓*Consistent Resource Archetype Names* vs. ✗*Inconsistent Resource Archetype Names*, ✓*Identifier Annotation* vs. ✗*Identifier Ambiguity* have significant differences in average TAU distribution. The other linguistic patterns and antipatterns pairs also have significant differences in average TAU distributions except for ✓*Contextual Resource Names* and ✗*Contextless Resource Names* patterns and antipatterns pair where average TAU distribution is similar.

Furthermore, for most linguistic design patterns and antipatterns, the TAU distributions are well-dispersed, tending to spread between 0 and 1, except for the ✗*Contextless Resource Names* and ✗*Identifier Ambiguity* antipattern. In those cases, TAU

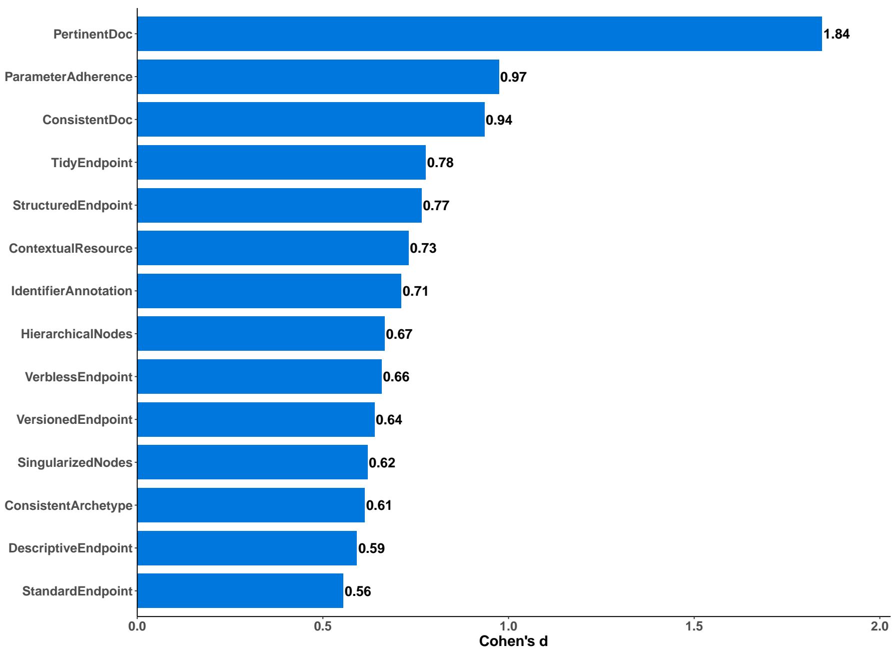

Figure 7.12: Effect sizes for differences in TAU between patterns and antipatterns, ranked by Cohen's d. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns.

values are clustered around 1 or 0.

We also performed hypothesis testing to confirm if the visually identified differences between linguistic design patterns and antipatterns were statistically substantial. We applied Holm-Bonferroni correction as we had multiple linguistic design patterns and antipatterns. The test result showed that all 14 linguistic design patterns and antipatterns pairs had a significant impact on understandability, i.e., the hypothesis tests produced significant results (See Table B.3).

Furthermore, we also calculated the effect size for the significant tests via Cohen's d. Figure 7.12 shows the d values generated for each of the 14 linguistic design patterns and antipatterns pairs. A detailed statistics are provided in Table B.3 in the appendix. Linguistic design patterns such as ✓*Pertinent Documentation* and its corresponding antipatterns produced a very large effect ($01.2 \leq d < 2.0$). ✓*Parameter Adherence*, ✓*Consistent Documentation* and their corresponding linguistic

antipatterns have a large effect ($0.8 \leq d < 1.2$). The rest of the linguistic design patterns and antipatterns pairs exhibit medium effect ($0.5 \leq d < 8$). All 14 linguistic patterns and antipatterns pairs have an effect higher than 0.5 (Cohen's $d \geq 5.0$). One pair has a very large effect ($d \geq 1.2$), two pairs have a large effect ($d \geq 0.8$), and 11 pairs have a medium effect ($d \geq 0.5$). All 14 pairs exhibit at least a medium effect, demonstrating that linguistic design patterns and antipatterns have an impact on the understandability of Web APIs.

RQ2.1 Summary: All 14 tasks following linguistic design patterns had better comprehension performance than those following linguistic design antipatterns. The effect sizes ranged from 0.56 to 1.84, with one linguistic design pattern and antipattern pairs having a Cohen's d greater than 1. Violating linguistic design patterns such as ✓*Pertinent Documentation*, ✓*Parameter Adherence*, and ✓*Consistent Documentation* had a strong negative impact on the understandability of APIs. Such a negative impact indicates that inconsistent documentation and a lack of alignment between an endpoint and its documentation can make it difficult for client developers to understand APIs.

RQ2.2: Perceived Difficulty in Understandability of Web APIs

From the RQ2.1, it is evident that all the tasks following linguistic design patterns significantly improve understandability. In RQ2, we aim to assess if linguistic design patterns and antipatterns significantly influence perceived difficulty in the understandability of APIs. Table 7.5 summarizes the results of the perceived difficulty ratings and correlates them with the corresponding TAU values. The mean and median difficulty ratings for understandability are lower for linguistic design patterns across all 14 tasks than linguistic design antipatterns. The difference in median rat-

Task	Median Rating		Mean Rating		Mean TAU	
	P	AP	P	AP	P	AP
Tidy Endpoint	2.0	3.0	2.07	3.31	0.7426	0.4017
Contextual Resource	2.0	3.0	1.80	3.03	0.8743	0.5692
Verbless Endpoint	2.0	3.0	1.82	3.17	0.6642	0.3702
Consistent Doc	2.0	3.0	1.91	3.13	0.7459	0.4393
Descriptive Endpoint	2.0	3.0	1.93	3.25	0.6684	0.4060
Hierarchical Nodes	2.0	3.5	1.78	3.19	0.6714	0.4081
Pertinent Doc	2.0	4.0	2.06	3.51	0.7696	0.1513
Standard Endpoint	2.0	3.0	1.94	3.25	0.7478	0.5041
Singularized Nodes	2.0	3.0	1.85	3.24	0.7192	0.5008
Versioned Endpoint	2.0	3.0	1.87	3.09	0.7214	0.5321
Parameter Adherence	2.0	4.0	2.04	3.43	0.8355	0.4631
Consistent Archetype	2.0	4.0	1.99	3.39	0.6099	0.3391
Identifier Annotation	2.0	3.0	2.00	3.26	0.7890	0.4677
Structured Endpoint	2.0	4.0	2.04	3.40	0.7705	0.4777

Table 7.5: Descriptive statistics for RQ2.2, perceived difficulty rating in understandability of APIs ranging from 1 (very easy) to 5 (very difficult), mean TAU provided for comparison.

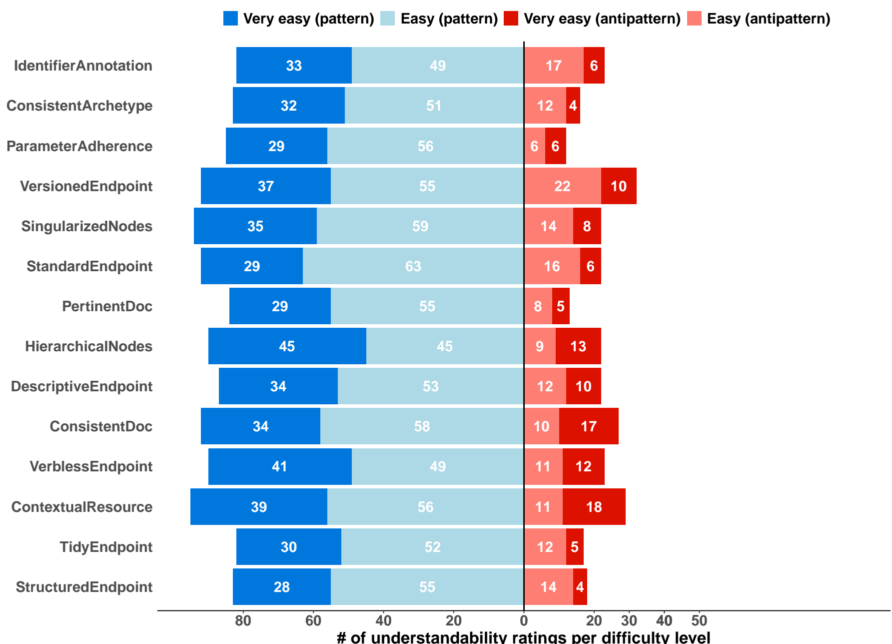

Figure 7.13: Bar plots of the perceived difficulty in understandability (RQ2.2) for the ratings 1 (very easy) and 2 (easy), linguistic design patterns ratings on the left, and linguistic design antipatterns ratings are on the right. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns.

ing is significant and ranges between 1 and 1.5 points. Significant differences in mean ratings and mean TAU values are also evident.

To further analyze the differences, we visualized the results using a comparative bar

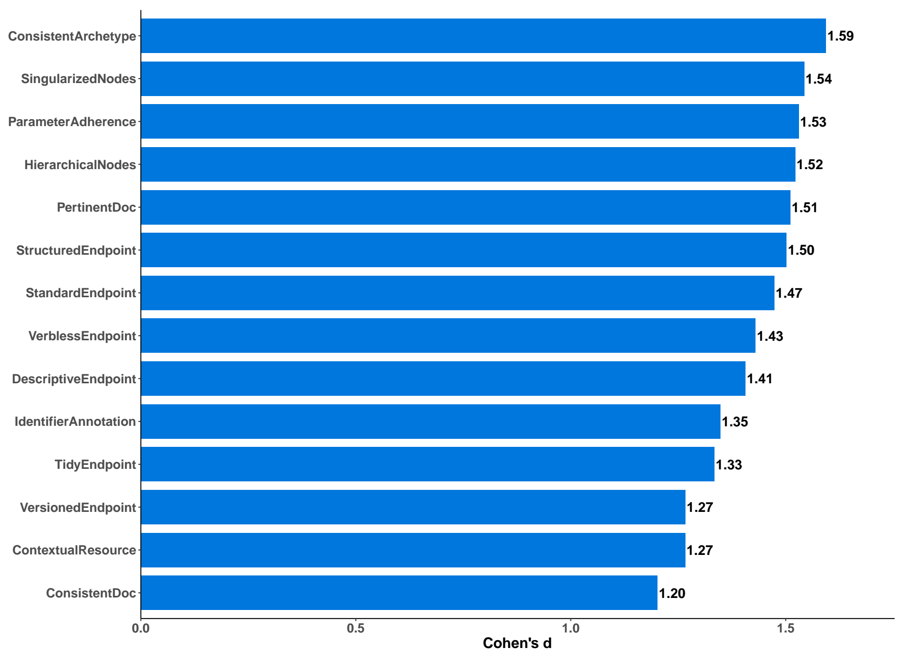

Figure 7.14: Effect sizes for differences in TAU between linguistic design patterns and antipatterns, ranked by Cohen's d for RQ2.2. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns.

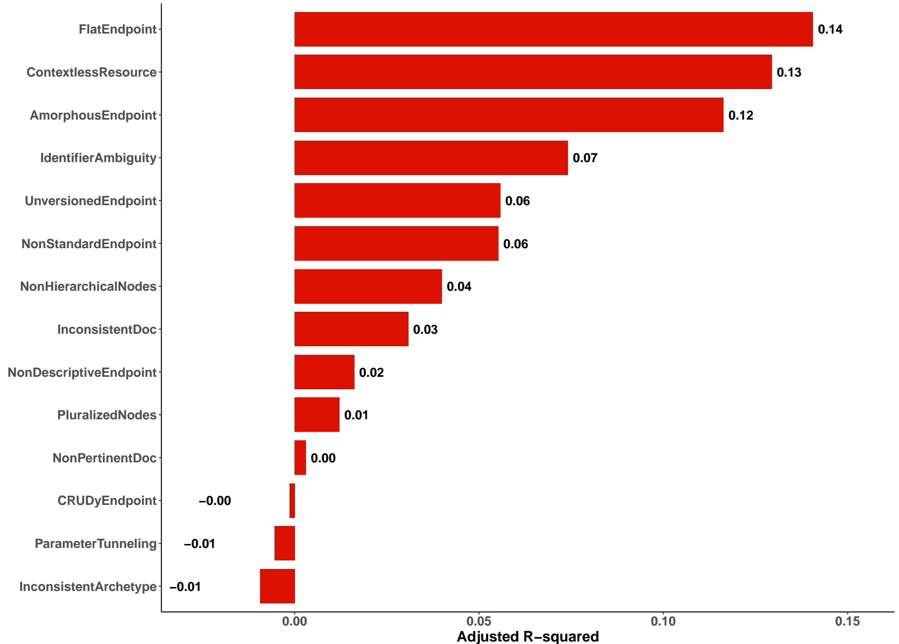

Figure 7.15: Adjusted R^2 values for the regression between TAU and perceived difficulty in understandability ratings for linguistic design antipatterns, ordered by adjusted R^2, higher is better.

plot of ratings 1 (very easy) and 2 (easy), as shown in Figure 7.13. The Figure further validates the difference between tasks following patterns and antipatterns. Out of 108 responses for *Identifier Annotation* task, 33 participants rated the tasks following patterns as *very easy*, and another 49 rated them as *easy* to understand (the remaining four participants selected ratings of 3, 4, or 5). In contrast, for tasks with *Identifier Ambiguity*, 17 participants rated them as *very easy*, and six participants rated them as *easy* to understand.

We also perform hypothesis testing to confirm the significant differences in perceived difficulty in understandability between tasks following linguistic design patterns and antipatterns. Figure 7.14 shows significant differences for all 14 tasks following linguistic patterns and antipatterns. Table B.4 in the appendix provides a detailed comparison. The effect sizes range from 1.20 to 1.59 (very large). ✓*Consistent Resource Archetypes Names*, ✓*Singularized Nodes*, ✓*Parameter Adherence*, ✓*Hierarchical Nodes*, ✓*Pertinent Documentation*, *Structured Endpoint* and their corresponding linguistic design antipattern have Cohen's $d > 1.5$. For the other eight linguistic design patterns and antipatterns pairs, the effect is greater than 1.2. Such values imply that violating these linguistic patterns has a strong negative impact on the understandability of APIs.

Violations of linguistic design patterns can also be identified by analyzing the correlation between TAU and the perceived difficulty in understanding API snippets following linguistic design antipatterns. Ideally, we aim to observe significant negative correlations between the two dependent variables (i.e., TAU and the perceived difficulty in understandability) for all API snippets following linguistic antipatterns. This indicates that the poorer the comprehension performance of snippets following antipatterns, the higher the perceived difficulty ratings should be. Snippets following antipatterns where this correlation does not hold may suggest that participants were confident in their performance despite responding incorrectly or taking longer,

potentially due to an unrecognized misunderstanding of the API snippet. For 12 patterns and antipatterns pairs, there was no significant negative correlation between TAU and perceived difficulty in understandability. In contrast, the ✓*Tidy Endpoint* and ✓*Structured Endpoint* patterns and antipatterns pairs have negative correlations. Detailed correlation results are available in the appendix in Table B.5. In Figure 7.15, we visualize the adjusted R^2 values for regression between TAU and the perceived difficulty rating for understandability for all 14 pairs of patterns and antipatterns. For most cases, the values are near zero, indicating that perceived difficulty ratings in understandability fail to explain any variation in TAU. This includes antipatterns with a significant impact on understandability, such as *Non-pertinent Documentation*, ✗*CRUDy Endpoint*, ✗*Parameter Tunneling*, and ✗*Inconsistent Resource Archetype Names*, making these antipatterns particularly concerning.

RQ2.2 Summary: For all 14 tasks, adherence to linguistic design patterns led to significantly lower perceived difficulty ratings for understandability than linguistic antipatterns. Effect sizes ranged from 1.20 to 1.59 (indicating very large effects). Furthermore, many tasks following linguistic design antipatterns showed no significant relationship between perceived difficulty ratings for understandability and TAU, highlighting their particularly problematic nature.

RQ2.3: Perceived Difficulty in Readability of Web APIs

In RQ2.3, we aim to assess if linguistic design patterns and antipatterns significantly influence perceived difficulty in the readability of APIs. Table 7.6 summarizes the results of the perceived difficulty ratings in readability and correlates them with the corresponding TAU values. The mean and median difficulty ratings for readability are lower for linguistic design patterns across all 14 tasks than linguistic design antipatterns. The difference in median rating is significant and ranges between 1

Task	**Median Rating**		**Mean Rating**		**Mean TAU**	
	P	**AP**	**P**	**AP**	**P**	**AP**
Tidy Endpoint	2.0	3.0	1.95	2.75	0.9691	0.4676
Contextual Resource	1.0	2.5	1.75	2.70	0.6974	0.4776
Verbless Endpoint	2.0	3.0	2.00	2.85	0.6435	0.2713
Consistent Doc	1.0	3.0	1.85	2.95	0.6255	0.3143
Descriptive Endpoint	2.0	3.0	1.85	2.90	0.4972	0.2874
Hierarchical Nodes	1.0	3.0	1.65	2.95	0.6676	0.4578
Pertinent Doc	1.5	3.5	1.90	3.30	0.7730	0.0942
Standard Endpoint	1.0	4.0	1.65	3.35	0.6221	0.3843
Singularized Nodes	1.0	3.0	1.85	3.10	0.6079	0.4024
Versioned Endpoint	1.0	3.0	1.70	2.75	0.7421	0.4445
Parameter Adherence	1.0	3.0	1.85	3.40	0.8835	0.2788
Consistent Archetype	2.0	3.0	1.85	3.00	0.4700	0.1788
Identifier Annotation	1.5	3.0	1.80	3.15	0.6447	0.4953
Structured Endpoint	1.0	4.0	1.80	3.80	0.8024	0.4847

Table 7.6: Descriptive statistics for RQ2.3, perceived difficulty rating in readability of APIs ranging from 1 (very easy) to 5 (very difficult), mean TAU provided for comparison.

and 3 points. We also observed significant differences in mean ratings and mean TAU values.

To further analyze the differences, we visualized the results using a comparative bar plot of ratings 1 (very easy) and 2 (easy). Figure 7.16 further shows the difference between tasks following patterns and antipatterns. For ✓*Identifier Annotation* task, out of 108 responses, 48 participants rated the tasks following patterns as *very easy*, and another 35 rated them as *easy* to read (the remaining 25 participants selected ratings of 3, 4, or 5). In contrast, for ✗*Identifier Ambiguity* task, only 16 participants rated them as *very easy*, and six participants rated them as *easy* to read.

We also perform hypothesis testing to confirm the significant differences in perceived difficulty in readability between tasks following linguistic design patterns and antipatterns. From Figure 7.17, we observed significant differences for all 14 tasks following linguistic design patterns and antipatterns. Table B.6 in the Appendix B provides a detailed comparison. The effect sizes (Cohen's d) range from 1.27 (medium) to 1.168 (very large). ✓*Singularized Nodes*, ✓*Standard Endpoint*, ✓*Pertinent Documentation*, ✓*Consistent Resource Archetype Names*, ✓*Structured Endpoint*

, and their corresponding antipattern have Cohen's $d \geq 1.5$, and the values range from 1.5 to .68. For all other linguistic design patterns and antipatterns pairs, the effect is greater than 1.27 (effect sizes are very large). All linguistic patterns and antipatterns pairs have d values greater than 1.2 (the effect is very large), which means violating these linguistic patterns has the strongest negative impact on the perceived difficulty in the readability of Web APIs.

We also analyzed the correlation between TAU and API snippets' perceived difficulty in readability following linguistic design antipatterns. Negative correlations between the two dependent variables (i.e., TAU and the perceived difficulty in readability) for all API snippets following linguistic design antipatterns are expected in an ideal scenario. This indicates that the poorer the comprehension performance of snippets following linguistic design antipatterns, the higher the perceived difficulty ratings in the readability of APIs should be. For 12 patterns and antipatterns pairs, there was no significant negative correlation between TAU and perceived difficulty in readability. In contrast, ✗*Contextless Resource Names* (Kendall's $\tau = -0.2870$), and ✗*Flat Endpoint* (Kendall's $\tau = -0.3523$) have negative correlations. Detailed correlation results are available in the Table B.7 in the appendix. Figure 7.15 shows the adjusted R^2 values for regression between TAU and the perceived difficulty rating for readability for all 14 pairs of patterns and antipatterns. For most patterns and antipatterns pairs, R^2 values are close to zero, indicating that perceived difficulty ratings in readability fail to explain any variation in TAU. The distribution of R^2 values suggests linguistic antipatterns such as ✗*CRUDy Endpoint*, ✗*Inconsistent Resource Archetype*, ✗*Parameter Tunneling* has a significant impact on the readability of Web APIs. Such a significant impact on the readability of APIs makes these linguistic design antipatterns especially dangerous.

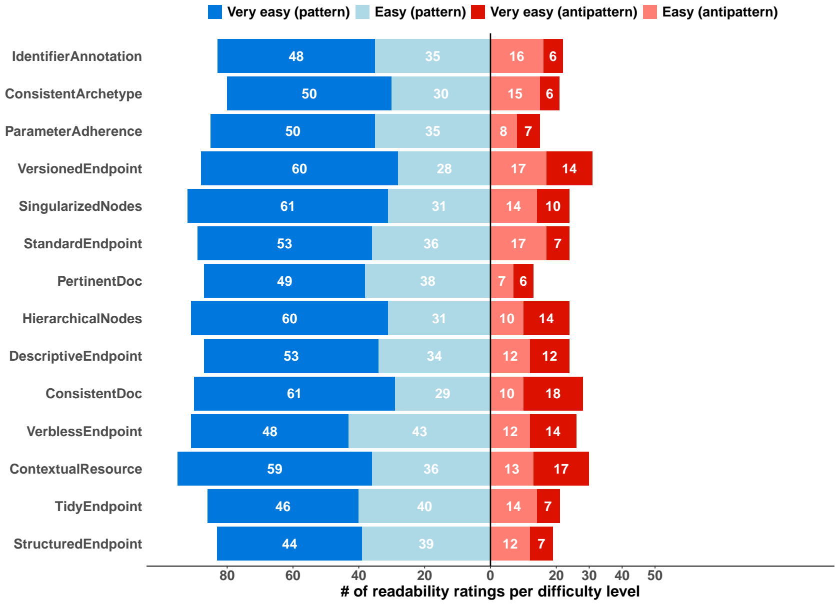

Figure 7.16: Bar plots of the perceived difficulty in readability (RQ2.3) for the ratings 1 (very easy) and 2 (easy), patterns ratings on the left, and antipatterns ratings on the right. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns.

> **RQ2.3 Summary:** For all 14 tasks, the following linguistic design patterns have significantly lower perceived difficulty ratings for readability compared to tasks that follow linguistic antipatterns. Cohen's d (effect sizes) ranged from 1.27 to 1.68, with tasks having very large effects (Cohen's $d \geq 1.2$). Moreover, many tasks following antipatterns exhibited a particularly problematic nature, as there is no significant relationship between perceived difficulty ratings for readability and TAU.

RQ2.4: Relationships with Demographic Attributes

In RQ2.4, we investigate whether there are any relationships between the dependent variables (TAU, perceived difficulty ratings in understandability, and readability) and the demographic attributes of our participants. We analyze this question separately for each task. We utilize predictors that included being from Canada, working in academia (vs. industry), being a student, being an API developer/designer, having

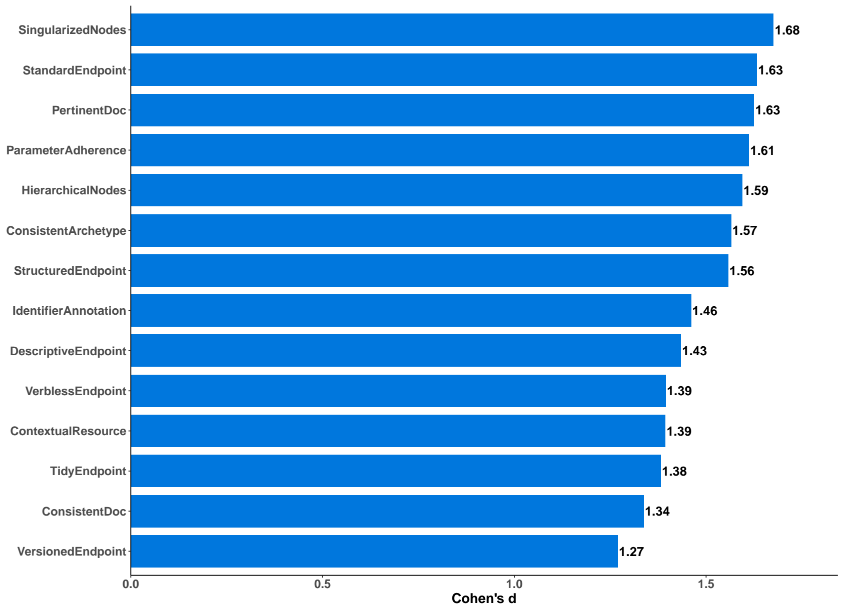

Figure 7.17: Effect sizes for differences in TAU between patterns and antipatterns, ranked by Cohen's d for RQ2.3. The Y-axis represents 14 linguistic design patterns and their corresponding antipatterns.

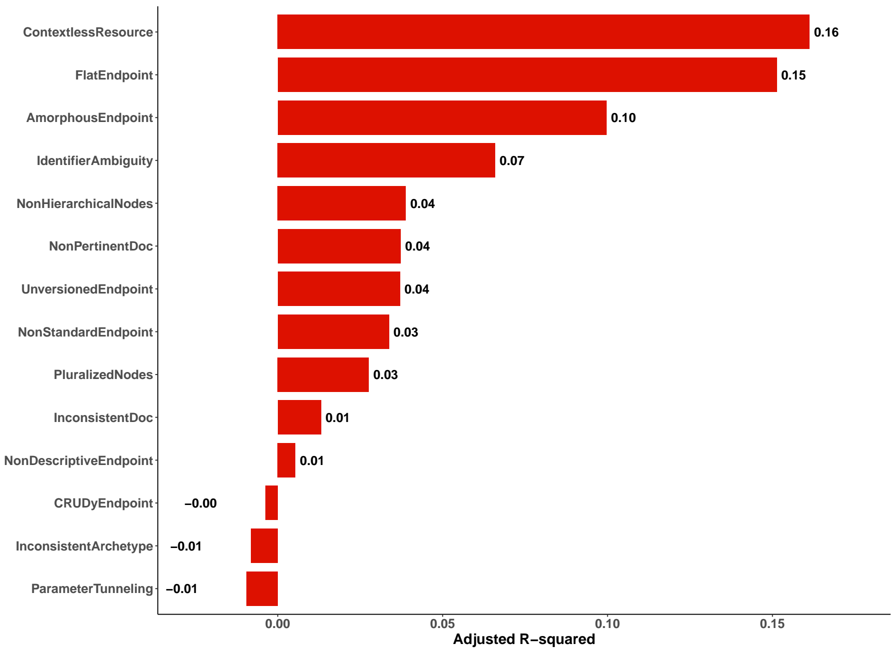

Figure 7.18: Adjusted R^2 values for the regression between TAU and perceived difficulty ratings in readability for antipattern for RQ2.3, ordered by adjusted R^2, higher is better.

years of API design experience, knowledge of the Richardson Maturity Model, and the preferred minimal maturity level. Although this was an exploratory research

question, we applied Holm-Bonferroni adjusted p-values with a significance level of $\alpha = 0.05$.

We did not find significant relationships from our analysis on perceived difficulty ratings in understandability and readability. *Years of API design experience* showed a small positive correlation with TAU for tasks following patterns (Kendall's $\tau = 0.0368, p = 0.6037$), indicating that participants with more experience tended to perform slightly better on API snippets adhering to patterns. However, we observed no positive correlations for API snippets following antipatterns. There was also a small positive correlation for participants with knowledge of the Richardson maturity model $\tau = 0.0409, p = 0.5856$). Such a small positive correlation implies that knowledge of *Richardson maturity* models helps identify tasks following linguistic patterns but does not really for tasks following linguistic antipatterns. After applying the Holm-Bonferroni adjustment, the original p-value ($p = .6037$) of *Years of Experience* in API design experience increases ($p = 1$). The p-value for knowledge of *Richardson maturity* model also increases after adjustments. As a result, the decisive factor in understandability and readability appears to be the "*years of experience*" in API design and knowledge of *Richardson maturity* model. None of the other demographic attributes showed any significant relationships.

The identified correlations are not significant. *Experience in API design* is associated with improved performance in understanding and reading APIs for tasks adhering to patterns. However, tasks following antipatterns are not significantly influenced by years of API design experience, as participants still found them difficult to understand and read. Participants with API design experience were more likely to sense that something was wrong in tasks with antipatterns, even if it did not enhance their understandability or readability. Similarly, knowledge of *Richardson maturity* model helps to understand and read tasks that follow linguistic patterns. However, knowledge of *Richardson maturity* model is not particularly helpful in cases where

tasks follow linguistic antipatterns.

We have also built linear regression models to predict TAU for tasks based on demographic attributes. The result also supports the theory that participants with experience and knowledge of the Richardson maturity model are more likely to notice something is wrong in tasks with antipatterns, but it does not help them with understandability or readability. Both models (patterns and antipatterns tasks) failed to provide reliable predictions for most of the sample, the model for tasks following patterns performed significantly worse. The antipattern model explained a reasonable percentage of variability (adjusted $R^2 = -0.0021$, $p = 0.4492$), whereas the patterns model had no explanatory power (adjusted $R^2 = 0.0084$, $p = 0.3231$).

RQ2.4 Summary: Among all the demographic attributes, only *years of experience* and knowledge of *Richardson maturity* model in API design have some degree of correlation with perceived difficulty in understandability and readability of Web APIs. However, *experience in API design* and knowledge of *Richardson maturity* model do not influence the tasks following linguistic design antipatterns, as participants find these tasks difficult to understand and read regardless of their experience with API design.

7.2.1 Discussions on Impact

Our survey results suggest that APIs following linguistic antipatterns have a negative impact on the understandability of APIs. In this section, we discuss the impact of some linguistic patterns and antipatterns on their understandability and readability.

✓*Contextualized Resource Names* linguistic patterns have a positive impact on the understandability and readability (see Figures 7.9, 7.13, and 7.16). In contrast, ✗*Contextless Resource Names* antipatterns have a negative impact on the understandability and readability of APIs. ✗*Contextless Resource Names* occurs when

nodes in an endpoint are not contextually related. The lack of contextual or semantic relation among the nodes of an endpoint makes it difficult for the client developer to comprehend and use the endpoint. The results of our impact study also support this statement and suggest using contextually related terms while designing API endpoints.

Similarly, ✓*Pertinent Documentation* linguistic patterns also have a positive impact on the understandability and readability of APIs (see Figure 7.9, 7.13, and 7.16). ✓*Pertinent Documentation* antipattern occurs when an endpoint and its corresponding documentation are semantically related (see Chapter 4). In contrast, ✗*Non-pertinent Documentation* occurs when the endpoint does not use semantically related terms to describe its purpose. The absence of a semantic relationship between the endpoint and its documentation reduces the understandability and readability of the endpoints. Findings from *RQ2.1, RQ2.2*, and *RQ2.3* also show the significance of ✓*Pertinent Documentation* on understandability and readability of APIs.

✗*Inconsistent Documentation* occurs when the HTTP method of an endpoint is in contradiction with its documentation. ✗*Inconsistent Documentation* reduces the understandability and readability of APIs (see Figure 7.9, 7.13, and 7.16). In contrast, ✓*Consistent Documentation* patterns enhance the understandability and readability of APIs.

Other linguistic antipatterns also reduce the understandability and readability of APIs. The discussion of research questions (RQs) shows that all 14 linguistic antipatterns are sensitive and severely impact APIs. So, API developers should avoid these antipatterns and adhere to linguistic patterns to design quality APIs that are easy for client developers to read and understand.

7.2.2 Implications for Developers

Client developers rely on API documentation to consume APIs effectively. Poorly designed APIs make reading and understanding difficult for client developers. The findings of our impact study indicate that linguistic antipatterns negatively affect the understandability and readability of APIs. Moreover, our results show that all 14 linguistic antipatterns are influential and significantly impact API comprehension.

API developers can use these insights to avoid incorporating linguistic antipatterns in their designs. Instead, they can incorporate linguistic patterns to improve API quality for client developers. Our findings suggest that ✗*Contextless Resource Names*, ✗*Non-pertinent Documentation*, and ✗*Inconsistent Documentation* have a particularly significant impact on API understandability and readability. Therefore, API developers should avoid them while designing APIs for clients. Avoiding linguistic antipatterns and incorporating linguistic patterns will enhance the understandability and readability of APIs. Furthermore, it will increase user adoption and improve the popularity of the API.

Chapter 8

Threats to Validity

This chapter discusses the threats to the validity of this thesis. In the following, we discuss our efforts to mitigate potential threats to the validity of our thesis.

8.1 External validity

External validity refers to the generalizability of findings to other environments and settings. We performed experiments using a dataset that included 4,027 endpoints from 69 REST and GraphQL APIs to minimize threats to the external validity of our findings. Analyzing these endpoints across various domains further strengthens the generalizability of our results.

To minimize the external validity of our impact study, we conducted our survey with 108 participants from diverse backgrounds. Participants had varying experiences with API design, including a mix of students and professionals from academia and industry. The number of participants is fairly substantial compared to standard software engineering experiments [74]. Our study produces statistical evidence of the effect of poor design practices on their understandability and readability.

The participants of our study were scattered across several countries, which reduced any country-specific biases in the experiment. We designed our API snippets using Swagger Editor to simulate real-world API documentation. Additionally, the API snippets were designed after real-world APIs. This approach minimized the threat of interaction effects with the API snippets.

In practice, developers often rely on supplementary materials, such as live API endpoints (for testing) and extended documentation. However, these were not incorporated into our study. Participants had to rely solely on the provided API snippets designed with the Swagger editor, which differs from a real-world environment. In practice, software engineers often access API documentation or interact with a running API instance for manual testing when using a web API. However, this limitation aligns with scenarios where such resources are unavailable, ensuring relevance to real-world constraints.

Furthermore, adding additional artifacts in API snippets could increase the participant's understandability and readability time. Therefore, we only provide the necessary information on API to help participants understand the purpose of an endpoint without having to analyze additional information. The results of our study are both transferable and reproducible in real-world environments while maintaining a strong degree of generalization. However, further research could be conducted in less controlled, real-world scenarios to explore the extent of this generalization.

8.2 Internal validity

Internal validity could be affected by threats that influence the dependent variables in this thesis. Our detection algorithms can identify linguistic antipatterns with an average accuracy of 94%. To minimize the internal validity of our detection results,

we used WordNet, LDA topic modeling, and the Cosine similarity metric for semantic analysis. The detection result may also vary based on how the heuristics of different patterns and antipatterns are defined and may vary across developers. Moreover, the definition of ground truth was conducted on 94 endpoints out of 4,027 analyzed endpoints, which may not represent the entire population. However, we opted for a confidence interval of 10 and a confidence level of 95% to minimize the threat. Thus, the ground truth was defined on 94 endpoints for ten antipatterns, i.e., $94 \times 14 = 1316$ questions.

To eliminate threats to the internal validity of our impact study, we followed a hybrid experimental design, incorporating both between-subjects and crossover approaches, reducing inter-participant variability, such as differences in API experience or professional background. Moreover, the relatively short experimental design minimizes the potential for history [74] and maturation effects [74]. Tasks were presented in randomized order, which helped reduce learning effects. However, randomization could introduce a risk of carryover effects. The study was conducted online, which limited our control over participants' environments and may have introduced distractions (e.g., background noise or interruptions). To address this, we excluded responses with implausible task durations (below 5 seconds or above 4 minutes) to ensure that participants remained focused.

8.3 Construct validity

Construct validity ensures that the experiment and its collected measures accurately represent the studied concepts. Accuracy and completeness of the collected dataset play a major role in detecting linguistic antipatterns. Anomalies in the dataset may impact the reliability of our results. Thus, efforts were made to ensure dataset quality during the data collection process. To minimize the construct validity, we

only collected well-structured and documented endpoints. We thoroughly analyzed linguistic patterns and antipatterns to define the detection heuristics and minimize construct validity. Additionally, how the three professionals decide on antipattern instances is subjective. This subjectivity could introduce potential biases in our detection performance. To reduce bias, three professionals defined ground truth, and majority voting was applied to decide on patterns and antipatterns.

In our impact study, we utilized TAU as the primary metric for measuring API understandability and readability [61]. TAU combines correctness and generated values ranging from 0 to 1 (higher values indicating better understandability). While TAU effectively combines multiple aspects (time, correctness) into a single value, it presents a non-ideal trade-off between time and correctness, particularly when correctness is binary. For example, while tasks with patterns like ✓*Parameter Adherence* achieved 90% correctness rate, their antipattern counterparts dropped to 51%, highlighting clear contrasts (See Figure 7.7). However, TAU considers both correctness and time to reduce such limitations. For perceived difficulty in understandability and readability, we used ordinal scales ranging from 1 to 5. The use of such ordinal scales is widely used and accepted for collecting subjecting ratings from participants [74]. We explicitly mention the objective of our study in the survey to avoid hypothesis guessing [74]. It is highly unlikely that participants would intentionally provide biased results for tasks following antipatterns. In our study, each linguistic pattern and antipattern represents a combination of several REST design rules [43, 42, 45], thereby minimizing the potential for mono-operational bias.

8.4 Reliability Validity

Reliability threats refer to factors that can affect the consistency and dependability of a study. To reduce the threat to the reliability of our detection method, we made

sure the collected data was accurate and consistent. Moreover, semantic and syntactic analysis was used to detect linguistic patterns and antipatterns in APIs. The incorporation of semantic and syntactic analysis helped in the accurate detection of linguistic patterns and antipatterns. Furthermore, we defined ground truth (validation survey) involving three software professionals with API design experience. Ground truth values were compared with the detection result to calculate accuracy, precision, recall, and F1 scores.

In our impact study, we use a state-of-the-art survey design approach [28, 75] to minimize threats to reliability. Furthermore, we created Web API snippets using the OpenAPI specification format to generate realistic API snippets that resemble real-world implementations. The survey included 108 participants from diverse professions and geographical locations. Proper cleaning methodology was used to filter our invalid and incomplete responses. Finally, we also performed several statistical analyses to test the defined hypothesis on the survey data.

To minimize further threats to validity and increase reliability and reproducibility, we published all resources related to this thesis at `https://github.com/krishnodey/API-Quality`.

Chapter 9

Conclusion and Future Works

This thesis evaluated the linguistic design quality of APIs of distributed systems and microservices. APIs act as the primary means of communication among services within these systems. The linguistic quality of APIs plays an important role in their adoption and use. APIs applying linguistic design patterns are easy to understand and adopt, while those with antipatterns hinder understanding and adoption. We analyzed 4,027 endpoints from 69 REST and GraphQL APIs. We implemented detection algorithms to perform semantic and syntactic analysis of endpoints to detect linguistic patterns and antipatterns. Our findings confirmed that linguistic antipatterns exist in APIs of distributed systems and microservices (RQ1.1). Our detection algorithms achieved an average accuracy of 94.07%, precision of 82.77%, recall of 87.35%, and F1-score of 88.06% (RQ1.2). Moreover, we observed that both REST and GraphQL APIs are prone to linguistic antipatterns, with REST having slightly more antipatterns (RQ1.3). Finally, ✗*Unversioned Endpoint*, ✗*Amorphous Endpoint*, and ✗*Pluralized Nodes* are the most commonly occurring antipatterns in APIs (RQ1.4).

In our impact study, we assessed the effect of API design practices on the under-

standability and readability of APIs. A controlled web experiment was conducted on 14 linguistic design patterns and antipatterns to assess their impact on understandability and readability [43, 42, 36]. We presented 28 API snippets (14 following linguistic patterns and 14 following linguistic design antipatterns) to the participants. Participants were asked to answer comprehension questions about API snippets and rate their understandability and readability. The experimental results demonstrate that adhering to linguistic design patterns (good design practices) significantly improves the understandability and readability of APIs. Snippets that followed linguistic design patterns were consistently easier for participants to understand and read, leading to higher correctness rates, less mean response time, and higher mean TAU scores than the tasks following linguistic antipatterns. In contrast, snippets following linguistic design antipatterns led to confusion and complexity, resulting in lower performance across all evaluation metrics, such as correctness, response time, and TAU score. The results emphasize the importance of clear, consistent, and well-structured design in enhancing the developer experience. The task following linguistic antipatterns had a negative impact on the understandability and readability of APIs. Additionally, participants with experience in API design performed better in tasks following linguistic design patterns. However, experience in API design does not reduce the perceived difficulty in understandability and readability of tasks (API snippets) following linguistic antipatterns.

As part of future work, further investigation is required to refine our detection algorithms to improve detection accuracy. We also plan to analyze a larger set of APIs and endpoints to further assess linguistic design quality. Furthermore, we aim to employ Large Language Models (LLMs) to improve detection accuracy, generate API design recommendations, and refactor linguistic antipatterns in APIs. Additionally, we plan to explore how LLMs can assist in generating API documentation and specifications from endpoints.

Our impact study provides valuable insights and opens several avenues for future research. Expanding the participant pool to include a broader and more diverse set of developers would offer deeper perspectives on how linguistic design patterns and antipatterns influence API understandability and readability.

While this thesis focused on specific linguistic patterns and antipatterns, similar experiments could be conducted on other API design rules and principles to further refine best practices. Beyond controlled experiments, evaluating the impact of linguistic design patterns in real-world software development projects would provide even stronger, practice-driven insights, bridging the gap between research and industry adoption.

Bibliography

[1] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus, *Lexicon Bad Smells in Software*, 2009 16th Working Conference on Reverse Engineering, IEEE, 2009, pp. 95–99.

[2] F Brito Abreu, Miguel Goulão, and Rita Esteves, *Toward the Design Quality Evaluation of Object-oriented Software Systems*, Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, 1995, pp. 44–57.

[3] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza, *A Large-Scale Empirical Study on Linguistic Antipatterns Affecting APIs*, 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 25–35.

[4] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol, *Linguistic Antipatterns: What they Are and How Developers Perceive them*, Empirical Software Engineering **21** (2016), 104–158.

[5] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc, *Repent: Analyzing the Nature of Identifier Renamings*, IEEE Transactions on Software Engineering **40** (2014), no. 5, 502–532.

[6] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli, *Automatic Synthesis of Behavior Protocols for Composable Web-services*, Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009, pp. 141–150.

[7] Kevin Bishop, Greg Bolan, David Bowen, Chris Cromack, Steve Evans, Raymond P Fisk, Walter Ganz, Mike Gregory, Robert Johnston, Jos Lemmink, et al., *Succeeding through Service Innovation: A Service Perspective for Education, Research, Business and Government*, University of Cambridge Institute for Manufacturing (2008), 1–30.

[8] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann, *Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality*, 2019 IEEE international conference on software architecture companion (ICSA-C), IEEE, 2019, pp. 187–195.

[9] Justus Bogner, Sebastian Kotstein, Daniel Abajirov, Timothy Ernst, and Manuel Merkel, *RESTRuler: Towards Automatically Identifying Violations of RESTful Design Rules in Web APIs*, 2024 IEEE 21st International Conference on Software Architecture (ICSA) (2024), 123–134.

[10] Justus Bogner, Sebastian Kotstein, and Timo Pfaff, *Do RESTful API Design Rules have an Impact on the Understandability of Web APIs?*, Empirical software engineering **28** (2023), no. 6, 132.

[11] Hayet Brabra, Achraf Mtibaa, Fabio Petrillo, Philippe Merle, Layth Sliman, Naouel Moha, Walid Gaaloul, Yann-Gaël Guéhéneuc, Boualem Benatallah, and Faïez Gargouri, *On Semantic Detection of Cloud API (Anti) Patterns*, Information and Software Technology **107** (2019), 65–82.

[12] Hayet Brabra, Achraf Mtibaa, Layth Sliman, Walid Gaaloul, Boualem Benatallah, and Faiez Gargouri, *Detecting Cloud (Anti) Patterns: OCCI Perspective*, Service-Oriented Computing: 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings 14, Springer, 2016, pp. 202–218.

[13] Jacob Cohen, *Statistical Power Analysis for the Behavioral Sciences*, routledge, 2013.

[14] Carlos Eduardo C Dantas and Marcelo A Maia, *Readability and Understandability Scores for Snippet Assessment: An Exploratory Study*, Anais do Workshop de Visualização, Evolução e Manutenção de Software (VEM) (2021), 40–50.

[15] Krishno Dey, Hung Cao, and Francis Palma, *Assessing the Linguistic Design Quality of APIs of Distributed Systems and Microservices*, 2024 34th International Conference on Collaborative Advances in Software and COmputiNg (CASCON), IEEE, 2024, pp. 1–10.

[16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina, *Microservices: yesterday, today, and tomorrow*, Present and ulterior software engineering (2017), 195–216.

[17] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Giuliano Antoniol, *Keep it Simple: Is Deep Learning Good for Linguistic Smell Detection?*, 2018 IEEE 25Th international conference on software analysis, evolution and reengineering (SANER), IEEE, 2018, pp. 602–611.

[18] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova, *Improving Source Code Readability: Theory and Practice*, 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE, 2019, pp. 2–12.

[19] Christiane Fellbaum, *WordNet: An Electronic Lexical Database*, MIT press, 1998.

[20] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000.

[21] Martin Fowler, *Richardson Maturity Model*, Online, 2010, Accessed: 2025-02-18.

[22] Amid Golmohammadi, Man Zhang, and Andrea Arcuri, *Testing RESTful APIs: A Survey*, ACM Transactions on Software Engineering and Methodology **33** (2023), no. 1, 1–41.

[23] Graphql.org, *Introduction to GraphQL*, Online, 2021.

[24] Florian Haupt, Frank Leymann, Anton Scherer, and Karolina Vukojevic-Haupt, *A Framework for the Structural Analysis of REST APIs*, 2017 IEEE International Conference on Software Architecture (ICSA), IEEE, 2017, pp. 55–58.

[25] Florian Haupt, Frank Leymann, and Karolina Vukojevic-Haupt, *API Governance Support Through the Structural Analysis of REST APIs*, Computer Science-Research and Development **33** (2018), 291–303.

[26] Michael Hausenblas, *On Entities in the Web of Data*, Springer New York, New York, NY, 2011.

[27] Daniel Jacobson, Greg Brail, and Dan Woods, *APIs: A Strategy Guide*, " O'Reilly Media, Inc.", 2012.

[28] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl, *Reporting Experiments in Software Engineering*, Guide to advanced empirical software engineering (2008), 201–228.

[29] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif, *An Empirical Study Assessing Source Code Readability in Comprehension*, 2019

IEEE International conference on software maintenance and evolution (ICSME), IEEE, 2019, pp. 513–523.

[30] M. G. Kendall, *A New Measure of Rank Correlation*, Biometrika **30** (1938), no. 1-2, 81–93.

[31] Ninus Khamis, René Witte, and Juergen Rilling, *Automatic Quality Assessment of Source Code Comments: the JavadocMiner*, Natural Language Processing and Information Systems: 15th International Conference on Applications of Natural Language to Information Systems, NLDB 2010, Cardiff, UK, June 23-25, 2010. Proceedings 15, Springer, 2010, pp. 68–79.

[32] Natalie Kiesler and Daniel Schiffner, *What is a Good API? A Survey on the Use and Design of Application Programming Interfaces*, International Conference on Internet of Everything, Springer, 2024, pp. 45–55.

[33] Peter Kolb, *DISCO: A Multilingual Database of Distributionally Similar Words*, Proceedings of KONVENS-2008, Berlin **156** (2008), 40–50.

[34] Sebastian Kotstein and Justus Bogner, *Which RESTful API Design Rules Are Important and How Do They Improve Software Quality? A Delphi Study with Industry Experts*, Service-Oriented Computing: 15th Symposium and Summer School, SummerSOC 2021, Virtual Event, September 13–17, 2021, Proceedings 15, Springer, 2021, pp. 154–173.

[35] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan, *Query Expansion via Wordnet for Effective Code Search*, 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 545–549.

[36] Mark Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*, " O'Reilly Media, Inc.", 2011.

[37] Cristian Mateos, Juan Manuel Rodriguez, and Alejandro Zunino, *A Tool to Improve Code-first Web Services Discoverability through Text Mining Techniques*, Software: Practice and Experience **45** (2015), no. 7, 925–948.

[38] Goran Mauša, Tihana Galinac-Grbac, and Bojana Dalbelo-Bašić, *A Systematic Data Collection Procedure for Software Defect Prediction*, Computer Science and Information Systems **13** (2016), no. 1, 173–197.

[39] Markus Neuhäuser, *Wilcoxon–Mann–Whitney Test*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[40] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino, *An Analysis of Public REST Web Service APIs*, IEEE Transactions on Services Computing **14** (2018), no. 4, 957–970.

[41] Sam Newman, *Building Microservices*, " O'Reilly Media, Inc.", 2021.

[42] Francis Palma, Javier Gonzalez-Huerta, Mohamed Founi, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc, *Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns*, International Journal of Cooperative Information Systems **26** (2017), no. 02, 1742001.

[43] Francis Palma, Javier Gonzalez-Huerta, Naouel Moha, Yann-Gaël Guéhéneuc, and Guy Tremblay, *Are RESTful APIs Well-designed? Detection of their Linguistic (Anti)Patterns*, Service-Oriented Computing: 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings 13, Springer, 2015, pp. 171–187.

[44] Francis Palma, Tobias Olsson, Anna Wingkvist, Fredrik Ahlgren, and Daniel Toll, *Investigating the Linguistic Design Quality of Public, Partner, and Private REST APIs*, 2022 IEEE International Conference on Services Computing (SCC), IEEE, 2022, pp. 20–30.

[45] Francis Palma, Tobias Olsson, Anna Wingkvist, and Javier Gonzalez-Huerta, *Assessing the linguistic quality of rest apis for iot applications*, Journal of Systems and Software **191** (2022), 111369.

[46] Francis Palma, Osama Zarraa, and Ahmad Sadia, *Are Developers Equally Concerned about Making their APIs RESTful and the Linguistic Quality? A Study on Google APIs*, Service-Oriented Computing: 19th International Conference, ICSOC 2021, Virtual Event, November 22–25, 2021, Proceedings 19, Springer, 2021, pp. 171–187.

[47] Luca Panziera and Flavio De Paoli, *A Framework for Self-descriptive RESTful Services*, Proceedings of the 22nd International Conference on World Wide Web, 2013, pp. 1407–1414.

[48] Allison Parrish, *Social network APIs: A Revised Lexical Analysis*, Online, 2010, [Accessed 14-01-2025].

[49] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann, *RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision*, Proceedings of the 17th international conference on World Wide Web, 2008, pp. 805–814.

[50] Fabio Petrillo, Philippe Merle, Francis Palma, Naouel Moha, and Yann-Gaël Guéhéneuc, *A Lexical and Semantical Analysis on REST Cloud Computing APIs*, Cloud Computing and Service Science: 7th International Conference, CLOSER 2017, Porto, Portugal, April 24–26, 2017, Revised Selected Papers 7, Springer, 2018, pp. 308–332.

[51] ———, *A Lexical and Semantical Analysis on REST Cloud Computing APIs*, Cloud Computing and Service Science: 7th International Conference, CLOSER 2017, Porto, Portugal, April 24–26, 2017, Revised Selected Papers 7, Springer, 2018, pp. 308–332.

[52] Petrillo, Fabio and Merle, Philippe and Moha, Naouel and Guéhéneuc, Yann-Gaël, *Are REST APIs for Cloud Computing Well-designed? An Exploratory Study*, Service-Oriented Computing: 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings 14, Springer, 2016, pp. 157–170.

[53] Antonio Quiña-Mera, Pablo Fernandez, José María García, and Antonio Ruiz-Cortés, *GraphQL: A Systematic Mapping Study*, ACM Computing Surveys **55** (2023), no. 10, 1–35.

[54] Mohammad Masudur Rahman and Chanchal K Roy, *TextRank Based Search Term Identification for Software Change Tasks*, 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 540–544.

[55] Faisal Rahutomo, Teruaki Kitasuka, and Masayoshi Aritsugi, *Semantic Cosine Similarity*, The 7th international student conference on advanced science and technology ICAST, vol. 4, 2012, p. 1.

[56] Dominik Renzel, Patrick Schlebusch, and Ralf Klamma, *Today's Top "RESTful" Services and Why They Are Not RESTful*, Web Information Systems Engineering-WISE 2012: 13th International Conference, Paphos, Cyprus, November 28-30, 2012. Proceedings 13, Springer, 2012, pp. 354–367.

[57] Leonard Richardson and Sam Ruby, *RESTful Web Services*, " O'Reilly Media, Inc.", 2008.

[58] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella, *REST APIs: A Large-scale Analysis of Compliance with Principles and best Practices*, Web Engineering:

16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings 16, Springer, 2016, pp. 21–39.

[59] Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo, *Improving Web Service Descriptions for Effective Service Discovery*, Science of Computer Programming **75** (2010), no. 11, 1001–1021.

[60] Shlomo S Sawilowsky, *New Effect Size Rules of Thumb*, Journal of modern applied statistical methods **8** (2009), 597–599.

[61] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto, *Automatically Assessing Code Understandability*, IEEE Transactions on Software Engineering **47** (2019), no. 3, 595–613.

[62] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto, *Improving Code Readability Models with Textual Features*, 2016 IEEE 24th International Conference on Program Comprehension (ICPC), IEEE, 2016, pp. 1–10.

[63] J P Shaffer, *Multiple Hypothesis Testing*, Annual Review of Psychology **46** (1995), no. Volume 46, 1995, 561–584.

[64] Unnati S Shah and Devesh C Jinwala, *Resolving Ambiguities in Natural Language Software Requirements: A Comprehensive Survey*, ACM SIGSOFT Software Engineering Notes **40** (2015), no. 5, 1–7.

[65] Samuel Sanford Shapiro and Martin B Wilk, *An Analysis of Variance Test for Normality (Complete Samples)*, Biometrika **52** (1965), no. 3-4, 591–611.

[66] Prabath Siriwardena, *Advanced API Security: OAuth 2.0 and Beyond*, Springer, 2020.

[67] Mark Steyvers and Tom Griffiths, *Probabilistic Topic Models*, Handbook of latent semantic analysis **427** (2007), no. 7, 424–440.

[68] Harihara Subramanian and Pethuru Raj, *Hands-On RESTful API Design Patterns and Best Practices: Design, Develop, and Deploy Highly Adaptable, Scalable, and Secure RESTful Web APIs*, Packt Publishing Ltd, 2019.

[69] Christoph Treude, Martin P Robillard, and Barthélémy Dagenais, *Extracting Development Tasks to Navigate Software Documentation*, IEEE Transactions on Software Engineering **41** (2014), no. 6, 565–581.

[70] Sira Vegas, Cecilia Apa, and Natalia Juristo, *Crossover Designs in Software Engineering Experiments: Benefits and Perils*, IEEE Transactions on Software Engineering **42** (2015), no. 2, 120–135.

[71] Xingwei Wang, Hong Zhao, and Jiakeng Zhu, *GRPC: A Communication Cooperation Mechanism in Distributed Systems*, ACM SIGOPS Operating Systems Review **27** (1993), no. 3, 75–86.

[72] Jim Webber, Savas Parastatidis, and Ian Robinson, *REST in Practice: Hypermedia and Systems Architecture*, " O'Reilly Media, Inc.", 2010.

[73] Fuguo Wei, Alistair Barros, and Chun Ouyang, *Deriving Artefact-Centric Interfaces for Overloaded Web Services*, Advanced Information Systems Engineering (Cham) (Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson, eds.), Springer International Publishing, 2015, pp. 501–516.

[74] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén, *Experimentation in Software Engineering*, Springer Berlin Heidelberg Springer, 2024.

[75] Marvin Wyrich, Justus Bogner, and Stefan Wagner, *40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study*, ACM Computing Surveys **56** (2023), no. 4, 1–42.

[76] Bernard P. Zeigler and Hessam S. Sarjoughian, *Service-Based Software Systems*, Springer London, London, 2013.

Appendix A

Impact Study Terminology

In this Chapter, we provide the terminologies used for the impact study. Then, we present current best practices for API design.

A.1 Terminology

This thesis focuses on resource-oriented Web APIs that use HTTP methods. We use the term Web API to describe any resource-oriented API that exposes functionality via Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URIs). A Web API is considered RESTful if it adheres to all major REST constraints defined by Fielding and Taylor [20]. There have been other studies that demonstrate the characteristics of RESTful API designs [36, 68, 57]. Studies in the literature suggest that following only some RESTful design constraints does not make an API RESTful. In the following, we define some of the terminologies that we use in the rest of this thesis.

REST: REST (Representational State Transfer) has become the de facto standard for designing APIs [20, 36]. REST APIs rely on HTTP (Hypertext Transfer Protocol)

requests and responses for their operation [20, 36, 68]. Throughout the thesis, we use the terms "Web APIs", "REST APIs" and "RESTful APIs" interchangeably.

Linguistic Design Patterns: APIs that adhere to good design practices and have good linguistic design are referred to as linguistic design patterns [43, 42]. A well-designed API (linguistic design patterns) provides the required functionality and interfaces for developers that are clear and easy to use [36, 72]. It also improves developer experience, reduces errors, eases maintenance, boosts integration, and increases easy adoption [36, 68]. We use the terms "linguistic design patterns", "linguistic patterns", and "patterns" interchangeably throughout the rest of the thesis.

Linguistic Design Antipatterns: APIs that do not adhere to good design practices and exhibit poor linguistic design are referred to as linguistic design antipatterns. A poorly designed API is hard to understand and use, lacks proper documentation, and increases development time for client developers [36, 68]. We use the terms "linguistic design antipatterns", "linguistic antipatterns", and "antipatterns" interchangeably throughout the rest of the thesis.

API Readability: The term "readability" refers to the ease with which one can read and comprehend an API endpoint [14]. It involves clarity, organization, and the use of intuitive naming conventions. A readable API enables developers to grasp its design without an unnecessary cognitive load.

API Understandability: The term "understandability" refers to the ease with which one can understand and interpret the functionality and purpose of an API endpoint [14, 10]. Understandability extends beyond just reading the API's syntax. It also includes how well the behaviour, usage, and purpose of APIs are conveyed, ensuring developers can integrate and utilize them effectively in their systems.

A.2 Best Practices in API Design

Several studies in the literature have tried to translate the REST constraints into more specific design guidelines in order to address the heterogeneity of interface designs among Web APIs. The purpose of these guidelines is to give developers practical advice on how to use HTTP and URI to create a consistent and efficient RESTful design. RESTful design rules, guidelines, and best practices are documented in many books in the literature [36, 68, 57, 72]. Studies such as Pautasso et al. [49], Palma et al. [43], Petrillo et al. [51], and Palma et al. [42] also highlight and propose the best RESTful design practices. Additionally, Martin Fowler defines a Richardson Maturity Model that enables web developers to estimate how well their web APIs comply with REST principles [21].

The model has four levels of maturity, which we shortly define in the following:

Level 0: Web APIs expose their functionality at the application level via a single URI and use the POST method to tunnel all the requests. SOAP and XML-RPC services belong to the level 0 in the Richardson Maturity Model.

Level 1: Web APIs that use different URIs to expose their resources (i.e., different URIs for different resources) but do not allow the use of different HTTP methods to perform different operations on their resources.

Level 2: Web APIs use different URIs for different resources and allow the use of HTTP to perform different operations on their resources. Level 2 Web APIs comply with the Uniform Interface restriction to some extent.

Level 3: Web APIs comply with the level 2 constraint and additionally also comply with the HATEOAS constraint ("Hypermedia As The Engine Of Application State"). The integration of HATEOAS improves the semantic relationship between resources

and their documentation.

Despite the existence of all these design guidelines, poor linguistic design is prevalent in REST APIs. In evaluating the linguistic design quality of APIs, researchers have identified both effective design practices (linguistic design patterns) and ineffective design practices (linguistic design antipatterns) [40, 58, 56, 43, 42, 45].

Appendix B

Impact Study Statistics

This chapter provides a detailed statistical analysis of our impact study.

Table B.1: Descriptive Statistics for RQ2.1. Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern. P: Linguistic design pattern, AP: Linguistic design antipattern.

Task	**Mean TAU**		**Mean Duration**		**#of Correct Answers**	
	P	**AP**	**P**	**AP**	**P**	**AP**
Tidy Endpoint	0.7426	0.4017	18.45	47.75	83 (77%)	47 (44%)
Contextual Resource	0.8743	0.5691	23.057	66.78	95 (88%)	63(58%)
Verbless Endpoint	0.6642	0.3702	20.83	25.84	76 (70%)	44 (41%)
Consistent Doc	0.7459	0.4393	18.12	22.82	101 (94%)	63 (58%)
Descriptive Endpoint	0.6683	0.4060	17.66	31.59	77 (71%)	49 (45%)
Hierarchical Nodes	0.6714	0.4081	23.83	27.47	86 (80%)	53 (49%)
Pertinent Doc	0.7696	0.1513	19.78	31.10	91 (84%)	19 (18%)
Standard Endpoint	0.7477	0.5041	21.10	25.77	85 (79%)	58 (54%)
Singularized Nodes	0.7192	0.5008	17.38	25.21	94 (87%)	69 (64%)
Versioned Endpoint	0.7214	0.5321	16.22	23.71	99 (92%)	83 (77%)
Parameter Adherence	0.8355	0.4631	15.44	26.23	97 (90%)	55 (51%)
Consistent Archetype	0.6099	0.3391	18.67	28.00	72 (67%)	40 (37%)
Identifier Annotation	0.7889	0.4677	20.55	49.55	86 (80%)	52 (48%)
Structured Endpoint	0.7705	0.4776	18.30	25.67	93 (86%)	61 (56%)

Table B.2: Extended Descriptive Statistics for TAU (RQ2.1). Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern. P: Linguistic design pattern, AP: Linguistic design antipattern.

Task	**Min TAU**		**Max TAU**		**Variance of TAU**	
	P	**AP**	**P**	**AP**	**P**	**AP**
Tidy Endpoint	0	0	0.9904	0.9844	0.1681	0.2165
Contextual Resource	0	0	0.9984	0.9981	0.1056	0.2426
Verbless Endpoint	0	0	0.9835	0.9827	0.1888	0.2100
Consistent Doc	0	0	0.9338	0.9264	0.0614	0.1534
Descriptive Endpoint	0	0	0.9816	0.9773	0.1824	0.2105
Hierarchical Nodes	0	0	0.9660	0.9589	0.1319	0.1802
Pertinent Doc	0	0	0.9762	0.9730	0.1152	0.1096
Standard Endpoint	0	0	0.9860	0.9855	0.1621	0.2229
Singularized Nodes	0	0	0.9469	0.9382	0.0898	0.1582
Versioned Endpoint	0	0	0.9266	0.9272	0.0612	0.1142
Parameter Adherence	0	0	0.9755	0.9739	0.0814	0.2108
Consistent Archetype	0	0	0.9754	0.9731	0.1907	0.1997
Identifier Annotation	0	0	0.9973	0.9968	0.1608	0.2468
Structured Endpoint	0	0	0.9709	0.9620	0.1017	0.1907

Table B.3: RQ2.1 Hypothesis testing for TAU, Holm-Bonferroni adjusted p-values with a significance level of $\alpha = 0.05$. The results are sorted by effect size (Cohen's d) to highlight the most impactful findings. Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern.

Task	**Test Statistics (U)**	**p-value**	**Cohen's d**	**Accepted**
Tidy Endpoint	8340	< 0.001	0.7772	yes
Contextual Resource	8027	< 0.001	0.7312	yes
Verbless Endpoint	7959.5	< 0.001	0.6582	yes
Consistent Doc	8568.5	< 0.001	0.9354	yes
Descriptive Endpoint	7671.5	< 0.001	0.5918	yes
Hierarchical Nodes	7993	< 0.001	0.6666	yes
Pertinent Doc	10075.5	< 0.001	1.8446	yes
Standard Endpoint	7823.5	< 0.001	0.5553	yes
Singularized Nodes	7519.5	< 0.001	0.6202	yes
Versioned Endpoint	7892	< 0.001	0.6392	yes
Parameter Adherence	8611	< 0.001	0.9741	yes
Consistent Archetype	7501.5	< 0.001	0.6129	yes
Identifier Annotation	7864.5	< 0.001	0.7115	yes
Structured Endpoint	8096.5	< 0.001	0.7658	yes
All rules combined	1568695	< 0.001	0.7551	yes

Table B.4: RQ2.2 Hypothesis testing for TAU, Holm-Bonferroni adjusted p-values with a significance level of $\alpha = 0.05$. The results are sorted by effect size (Cohen's d). Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern.

Task	**Test Statistics (U)**	**p-value**	**Cohen's d**	**Accepted**
Tidy Endpoint	9587.0	¡ 0.001	1.3342	yes
Contextual Resource	9270.0	¡ 0.001	1.2662	yes
Verbless Endpoint	9669.5	¡ 0.001	1.4299	yes
Consistent Doc	9162.0	¡ 0.001	1.2020	yes
Descriptive Endpoint	9643.0	¡ 0.001	1.4065	yes
Hierarchical Nodes	9788.0	¡ 0.001	1.5221	yes
Pertinent Doc	9916.5	¡ 0.001	1.5111	yes
Standard Endpoint	9782.5	¡ 0.001	1.4739	yes
Singularized Nodes	9909.0	¡ 0.001	1.5431	yes
Versioned Endpoint	9355.5	¡ 0.001	1.2666	yes
Parameter Adherence	9951.5	¡ 0.001	1.5304	yes
Consistent Archetype	9987.5	¡ 0.001	1.5939	yes
Identifier Annotation	9566.0	¡ 0.001	1.3480	yes
Structured Endpoint	9860.5	¡ 0.001	1.5020	yes

Table B.5: Correlation between perceived difficulty in understandability and TAU for RQ2.2. Values are sorted by Kendell's τ (correlation strength), Holm-Bonferroni adjusted p-values, $\alpha = 0.05$. Insignificant correlations are marked with (*).

Task	**Kendall's τ**	**p-value**	**Median Difficulty**		**Mean TAU**	
			P	**AP**	**P**	**AP**
Amorphous Endpoint	-0.2768	0.0044	2.0	3.0	0.7426	0.4017
Contextless Resource	-0.2407	0.0111*	2.0	3.0	0.8743	0.5692
CRUDy Endpoint	0.1097	1.0000*	2.0	3.0	0.6642	0.3702
Inconsistent Doc	-0.1457	0.1823*	2.0	3.0	0.7459	0.4393
NonDescriptive Endpoint	-0.1597	0.1823*	2.0	3.0	0.6684	0.4060
NonHierarchical Nodes	-0.1572	0.1823*	2.0	3.5	0.6714	0.4081
NonPertinent Doc	0.0863	1.0000*	2.0	4.0	0.7696	0.1513
NonStandard Endpoint	-0.2310	0.0178*	2.0	3.0	0.7478	0.5041
Pluralized Nodes	-0.1151	0.3327*	2.0	3.0	0.7192	0.5008
Unversioned Endpoint	-0.1542	0.1713*	2.0	3.0	0.7214	0.5321
Parameter Tunneling	-0.0698	0.7676*	2.0	4.0	0.8355	0.4631
Inconsistent Archetype	-0.0352	1.0000*	2.0	4.0	0.6099	0.3391
Identifier Ambiguity	-0.2305	0.0189*	2.0	3.0	0.7890	0.4677
Flat Endpoint	-0.3627	¡ 0.001	2.0	4.0	0.7705	0.4777

Table B.6: RQ2.3 Hypothesis testing for TAU, Holm-Bonferroni adjusted p-values with a significance level of $\alpha = 0.05$. The results are sorted by effect size (Cohen's d). Only linguistic pattern names are mentioned under the task column, but each task corresponds to both the pattern and its respective antipattern.

Tasks	**Test Statistics (U)**	**p-value**	**Cohen's d**	**Accepted**
Tidy Endpoint	9656.5	¡ 0.001	1.3816	yes
Contextual Resource	9561.0	¡ 0.001	1.3936	yes
Verbless Endpoint	9575.0	¡ 0.001	1.3945	yes
Consistent Doc	9441.5	¡ 0.001	1.3371	yes
Descriptive Endpoint	9685.0	¡ 0.001	1.4343	yes
Hierarchical Nodes	9889.0	¡ 0.001	1.5943	yes
Pertinent Doc	10043.0	¡ 0.001	1.6250	yes
Standard Endpoint	10012.0	¡ 0.001	1.6324	yes
Singularized Nodes	10077.0	¡ 0.001	1.6760	yes
Versioned Endpoint	9389.0	¡ 0.001	1.2697	yes
Parameter Adherence	10006.5	¡ 0.001	1.6117	yes
Consistent Archetype	9914.5	¡ 0.001	1.5653	yes
Identifier Annotation	9749.5	¡ 0.001	1.4610	yes
Structured Endpoint	9910.0	¡ 0.001	1.5579	yes

Table B.7: Correlation between perceived difficulty in understandability and TAU for RQ2.3. Values are sorted by Kendell's τ (correlation strength), Holm-Bonferroni adjusted p-values, $\alpha = 0.05$. Insignificant correlations are marked with (*).

Task	**Kendall's τ**	**p-value**	**Median Difficulty**		**Mean TAU**	
			P	**AP**	**P**	**AP**
Flat Endpoint	-0.2284	0.0243*	2.0	3.0	0.9691	0.4676
Contextless Resource	-0.2870	0.0011	1.0	2.5	0.6974	0.4776
CRUDy Endpoint	0.0762	1.0000*	2.0	3.0	0.6435	0.2713
Inconsistent Doc	-0.0920	0.5666*	1.0	3.0	0.6255	0.3143
NonDescriptive Endpoint	-0.1150	0.4612*	2.0	3.0	0.4972	0.2874
NonHierarchical Nodes	-0.1421	0.2713*	1.0	3.0	0.6676	0.4578
NonPertinent Doc	0.1987	1.0000*	1.5	3.5	0.7730	0.0942
NonStandard Endpoint	-0.1458	0.2644*	1.0	4.0	0.6221	0.3843
Pluralized Nodes	-0.1633	0.1521*	1.0	3.0	0.6079	0.4024
Unversioned Endpoint	-0.1110	0.4612*	1.0	3.0	0.7421	0.4445
Parameter Tunneling	-0.0357	0.9744*	1.0	3.0	0.8835	0.2788
Inconsistent Archetype	-0.0741	0.7198*	2.0	3.0	0.4700	0.1788
Identifier Ambiguity	-0.1784	0.1288*	1.5	3.0	0.6447	0.4953
Flat Endpoint	-0.3523	¡ 0.001	1.0	4.0	0.8024	0.4847

Vita

Candidate's full name: Krishno Dey

University Attended:

- Master of Computer Science, University of New Brunswick, Fredericton, NB, Canada (2023-2025)
- BSc in Computer Science and Engineering, Daffodil International University, Dhaka, Bangladesh, (2017-2021)

Publications:

- Journal:
  - **Dey, K.**, Cao, H., Thiel S., & Palma, F. *Linguistic Patterns and Antipatterns Detection and their Impact on Understandability and Readability of APIs*, **Journal of Software: Practice and Experience** (*Submitted in February 2025, Under Review*).
- Conference:
  - **Dey, K.**, Cao, H., & Palma, F. (2024, November). *Assessing the Linguistic Design Quality of APIs of Distributed Systems and Microservices.* In **34th International Conference on Collaborative Advances in**

Software and COmputiNg (CASCON) (pp. 1-10), November 2024, Toronto, ON, Canada.

- Workshop/Symposium:
  - **Dey, K.**, Cao, H. & Palma, F. (2024). *Syntactic and Semantic Analysis of REST and GraphQL APIs to Assess and Compare their Linguistic Design Quality* [Extended Abstract]. In The **30th UNB Annual Graduate Research Conference**. Fredericton, NB, Canada.
  - **Dey, K.**, Cao, H. & Palma, F. (2024). *Syntactic and Semantic Analysis of REST and GraphQL APIs to Assess and Compare their Linguistic Design Quality* [Poster]. In The **30th UNB Annual Graduate Research Conference**. Fredericton, NB, Canada.
  - **Dey, K.**, Cao, H., & Palma, F. (2024). *Semantic Analysis of REST and GraphQL APIs to Assess Linguistic Design Quality* [Poster], In **2024 Research Expo, Faculty Computer Science**, Fredericton, NB, Canada.

Conference Presentations:

- 35th IEEE International Conference on Collaborative Advances in Software and Computing (CASCON), November 2024, Toronto, ON, Canada.