Assessing the Linguistic Design Quality of APIs of Distributed Systems and Microservices

Krishno Dey*, Hung Cao†, Francis Palma*
* *SE+AI Research Lab, Faculty of Computer Science, University of New Brunswick, NB, Canada*
† *Analytics Everywhere Lab, Faculty of Computer Science, University of New Brunswick, NB, Canada*
{krishno.dey, hcao3, francis.palma}@unb.ca

***Abstract*—APIs or Application Programming Interfaces help distributed systems and microservices to expose their functionalities and serve as a means for communication among them. In contrast to a poorly designed API, a well-designed API is easy for users to understand and use. Thus, APIs with high-quality design are essential both for API providers and client developers. This paper aims to assess the linguistic design quality of APIs in distributed systems and microservices by automatically detecting good and poor design practices, commonly known as patterns and antipatterns, respectively. We rely on syntactic and semantic analyses for automatic assessment of the design quality of APIs using detection heuristics. Syntactic analysis involves analyzing the structure and syntax of the APIs, while semantic analysis involves analyzing API documentation, descriptions, and parameters. We achieved an overall accuracy of more than 93% in detecting patterns and antipatterns. Our detection results also suggest that antipatterns are prevalent in the APIs of distributed systems and microservices. Our findings will assist API developers in identifying poor design practices and improving the design quality of their APIs.**

***Index Terms*—Distributed Systems, Microservices, APIs, REST, GraphQL, Semantic Analysis, NLP, Linguistic Design Quality**

I. Introduction

Distributed systems and microservices work together to achieve a common (technical and/or business) objective. Nowadays, microservices are a very popular approach for building such distributed systems [1]. Application Programming Interfaces (APIs) facilitate communications among the (micro)services. APIs define the protocols, data formats, and operations for communication [2]. Moreover, distributed systems and microservices rely on APIs and their documentation to publish their functionalities. APIs may follow good design practices, i.e., design patterns, that exhibit high-quality design. On the other hand, APIs may also exhibit poor design practices, i.e., antipatterns.

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 [2]. In contrast, a poorly designed API is hard to understand and use, lacks proper documentation, and increases development time for client developers. Thus, automatic detection of poor linguistic design in APIs is essential for developing user-friendly and efficient APIs for distributed systems and microservices. Adopting good API design practices and early detection of poor design practices can significantly contribute to the success of an API.

In this study, we assess the linguistic design quality of the APIs of distributed systems and microservices. We aim to assess how well APIs expose their functionality. Various types of APIs are available, such as SOAP, REST, GraphQL, gRPC, etc. This study 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. 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 [3]. 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 [4]. Both REST and GraphQL follow a resource-oriented architecture [2], [4]. Every resource has its unique identifier, URI (Unique Resource Identifier), and a set of actions (HTTP methods) that the client can perform to manipulate resources.

This study aims to find empirical evidence of linguistic antipatterns in APIs of distributed systems and microservices, and identify the most prevalent linguistic antipatterns. The findings of our study will provide valuable insights for API developers, guiding them on the specific antipatterns that require attention to enhance the overall design quality of their APIs. Thus, we aim to answer the following research questions:

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

RQ2: What is the accuracy of the detection heuristics of linguistic patterns and antipatterns?

RQ3: Which API category in distributed systems and microservices is more prone to poor linguistic design?

RQ4: Which linguistic patterns and antipatterns are most common in APIs of distributed systems and microservices?

As our main contributions in this study, we:

- assess the linguistic design quality of APIs of distributed systems and microservices using syntactic and semantic analyses;

- provide empirical evidence that both REST and GraphQL APIs have poor linguistic design;
- analysis of 1,655 endpoints from 21 REST and 12 GraphQL APIs after collecting 846 endpoints from GraphQL APIs and 809 endpoints from REST APIs; and,
- identify most occurring patterns and antipatterns among the analyzed APIs.

The rest of the paper is organized as follows: Section II highlights the relevant studies from the literature on detecting linguistic antipatterns in APIs. Section III presents a brief definition of ten linguistic antipatterns from the literature. Section IV outlines our research methodology. Section V summarizes the detection results and answers the research questions, while Section VI presents a general discussion of our findings. Finally, Section VII concludes our discussion, highlights the limitations of our study, and proposes potential future work.

II. Related Work

This study aims to find the linguistic patterns and antipatterns in the APIs of distributed systems and microservices. Identifying patterns (i.e., good practices) and antipatterns (i.e., poor practices) are essential to assess the linguistic quality of APIs. We use syntactic and semantic analysis of the endpoints, description, and their parameters to assess the linguistic quality of APIs. REST and GraphQL APIs are two of the most widely used API design practices. Due to their flexibility and efficiency, developers of distributed systems and services follow them to design their APIs. In this study, we focus on the syntactic and semantic analysis of REST and GraphQL APIs to determine their linguistic quality. In the following, we summarize and present the previous works that use semantic and syntactic analysis to assess the linguistic quality of APIs.

Analysis of REST APIs: The linguistic design of REST APIs has been evaluated in several studies applying semantic analysis. A survey by Petrillo et al. [5] presented 73 best practices in REST API design. Other studies that detected bad practices in REST API design include Hausenblas et al. [6], which assessed endpoint naming quality by performing a subjective analysis on REST APIs. Parrish et al. [7] used subjective lexical comparison to examine how verbs and nouns are used in naming endpoints. To evaluate the extent to which developers follow the standard API practices, Rodrıguez et al. [8] analyzed the HTTP requests and found that REST APIs usually do not follow the standard design practices. However, those studies do not focus on the linguistic design quality of APIs.

Concerning linguistic quality, some of the most notable studies to detect linguistic patterns and antipatterns in APIs using syntactic and semantic analyses were performed using DOLAR [9] and SARA [10] tools, where the authors defined nine antipatterns and conducted their study on widely known REST APIs including Facebook and YouTube. Later, they studied APIs for IoT applications [11], analyzed 1,102 endpoints from 19 IoT APIs, and detected nine linguistic patterns and antipatterns. In another work, the authors conducted a comparative study on 37 public, partner, and private APIs [12] to investigate which type of API is more prone to poor linguistic design. Other works on REST APIs involved structural analysis of linguistic designs of APIs [13], [14]. Besides that, several surveys on API design quality [15], [16] provided a state-the-art summary of the current good and bad practices in API design. Semantic analysis is not limited to linguistic antipattern detection. For example, Ma et al. [17] used semantic analysis to develop a tool to help developers analyze, cluster, and recommend APIs.

Analysis of GraphQL APIs: Compared to REST, there has not been much work for GraphQL APIs in assessing linguistic quality. Most of the study on GraphQL is focused on optimizing GraphQL query processing [18], [19]. Wittern et al. [18] analyzed common naming conventions and worst-case response sizes, and described practices that address large responses. Roksela et al. [19] evaluated different query execution plans of GraphQL and suggested a selection of execution strategies to improve the performance of GraphQL APIs. Other studies, such as [20], focused on improving the performance optimization of GraphQL APIs. Finally, studies such as [21], [22] highlight the good and bad practices in GraphQL APIs. Interestingly, no study in the literature focused on the linguistic quality of GraphQL APIs.

This study seeks empirical evidence of poor linguistic design in REST and GraphQL. We also aim to conduct a comparative study between REST and GraphQL regarding their linguistic design quality.

III. Linguistic Patterns and Antipatterns in APIs

This section briefly defines linguistic patterns and antipatterns extracted from the literature. In the following, we briefly define all ten patterns and antipatterns used in this study.

✗Amorphous vs. ✓Tidy Endpoint: An endpoint is considered tidy if it has 1) lower-case resource naming, 2) no underscores, 3) no trailing slashes, and 4) no file extensions. *✗Amorphous Endpoint* antipattern, on the other hand, appears when endpoints have capital letters (except for camel cases) or other symbols that make them difficult to use and read [9]. For 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.

✗Contextless vs. ✓Contextualized Resource Names: Nodes in the endpoint should belong to the same semantic context, i.e., the endpoint should be contextual. *✗Contextless Resource Names* antipattern occurs when nodes in the endpoint do not belong to the same semantic context [9]. For example, endpoint /newspapers/earth/players/{id} is considered to be a *✗Contextless Resource Names* antipattern as nodes are not semantically related. In contrast, endpoint /football/club/players/{id} is considered to be a *✓Contextualized Resource Names* pattern as all the nodes are semantically related.

✗**CRUDy vs. ✓Verbless Endpoint:** ✓*Verbless Endpoint* does not use CRUDy terms such as create, read, update, delete, or their synonyms with HTTP methods. In contrast, using such terms as resource names is considered to be ✗*CRUDy Endpoint* [9]. The endpoint update/players/{id} along with the POST method is a ✗*CRUDy Endpoint* antipattern as it has CRUDy term *update*. In contrast, endpoint /players/{id} along with the POST method is a ✓*Verbless Endpoint* pattern as the endpoint does not contain any CRUDy terms or their synonyms.

✗**Inconsistent vs. ✓Consistent Documentation:** ✗*Inconsistent Documentation* antipattern occurs when the HTTP method of an endpoint is in contradiction with its documentation. In contrast, for consistent documentation, the HTTP method is in agreement with the documentation [11]. 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 [2]. 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.

✗**Non-descriptive vs. ✓Self-descriptive Endpoint:** In API design architecture, endpoints have to be as user-friendly as possible. An endpoint needs to be easy to understand and as precise as possible for better understandability. 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 grasp. A ✓*Self-descriptive Endpoint*, on the other hand, has resource identifiers that are short and to the point [12]. 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.

✗**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 [9]. For example, /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.

✗**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., in endpoint, the presence of blank spaces in endpoint, the usage of double hyphens in endpoint, and the presence of unknown characters in endpoint are the four main indicators of ✗*Non-standard Endpoint* design [11]. 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.

✗**Non-pertinent vs. ✓Pertinent Documentation:** ✗*Non-pertinent Documentation* antipattern occurs when an endpoint is in contradiction with its documentation, i.e., the endpoint and its corresponding documentation are not semantically related. In contrast, a properly documented endpoint uses semantically related terms to clearly describe its purpose [10]. In PokéAPI, the endpoint /v2/berry-firmness/{names}/ is in contradiction with the documentation *'The name of this resource is listed in different languages'*, and thus is a ✗*Non-pertinent Documentation*. In contrast, another endpoint documentation pairs from PokéAPI: /v2/berry-firmness/{berries}/ - *'A list of the berries with this firmness.'* shows a higher semantic relationship, and thus is a ✓*Pertinent Documentation*.

✗**Pluralized vs. ✓Singularized Nodes:** Endpoints should not use singular/plural nouns inconsistently when naming resources in the API. The last node of the request endpoint should be singular when clients send PUT/DELETE requests. 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 [9]. In Adobe Audience Manager API, the POST method is used with the /data-feeds/usage endpoint whose last node is a singular noun, and thus it is considered to be ✗*Pluralized Nodes* antipattern. In contrast, if PUT or DELETE were used with the same endpoint, it would have been considered to be ✓*Singularized* Nodes pattern, as singular last nodes are supposed to be used with PUT or DELETE methods.

✗**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 [11]. For example, the endpoint /file_requests/count from Dropbox is an ✗*Unversioned Endpoint* as the endpoint does not contain any version number. In contrast, another endpoint /v1/me/library/playlists/{id} from Apple Music is considered to be a ✓*Versioned Endpoint*, as the endpoint contains the version number.

Apart from these patterns and antipatterns, many other REST API design rules are defined in several books such as [2]. However, they do not discuss detection techniques or algorithms for these rules. This study analyzes ten patterns and antipatterns for API design from the literature.

IV. Research Methodology

Figure 1 shows the research methodology followed in this study. We collect and pre-process API endpoints in Steps

TABLE I: List of the analyzed APIs, online documentation, and number of endpoints.

REST				GraphQL	
APIs	Endpoints	APIs	Endpoints	APIs	Endpoints
Adobe Audience	65	Microsoft Power BI	34	AniList	27
Apple App Store	32	Node-RED	15	Apple Music	99
BroadCom	40	Oracle Cloud Marketplace	43	Artsy	21
CiscoFlare	25	QuickBooks Online	21	Braintree	96
ClearBlade	45	Samsung ARTIK Cloud	80	Facebook	66
Dropbox	32	Shopify	71	GitHub	256
Google Nest	35	SurveyJS	14	GitLab	55
GroupWise	56	Uber	24	Instagram	28
IBM Cloud Pak	34	WM3 Multishop	54	Pipefy	90
IBM Watson IoT	57			PokéAPI	24
Instagram	19			Shopify	33
Linkedin	13			Twitter	51

1 and 2. We detect linguistic patterns and antipatterns in Step 3 by implementing their detection heuristics. We define the ground truth in Step 4. Finally, we compute detection performance metrics, synthesize our findings, and answer our research questions in Step 5. The following sections provide brief descriptions of each step of the methodology.

A. Step 1: Data Collection

This step involves gathering information of APIs. For each API, we manually collected endpoints, associated HTTP methods, available documentation, and parameters. To ensure the data quality, we followed a systematic data collection approach [23]. Furthermore, for our investigation, we only gathered endpoints that have well-organized HTTP methods and documentation. The resulting dataset encompasses 846 endpoints from 12 GraphQL APIs. Additionally, we incorporate 809 endpoints from 21 REST APIs made publicly available in [12] to analyze the design quality of APIs. Table I lists 33 analyzed REST and GraphQL APIs with the sources of their online documentation and the number of endpoints analyzed.

B. Step 2: Pre-processing

We pre-process endpoint documentation and their parameters using standard NLP techniques before using them for the definition of ground truth and detection of patterns and antipatterns. As part of the pre-processing, we first refine endpoint documentation by removing extra spaces, unknown characters, unknown symbols, and non-English characters. Then, we expand acronyms and decompose compound words to improve the readability and understandability of endpoint documentation and parameters. From the APIs, we gather a list of acronyms and compound words and their corresponding split words for pre-processing purposes. Further pre-processing, such as stop word removal, stemming, and lemmatization are applied during the detection phase in Step 3.

C. Step 3: Detection of Linguistic Patterns and Antipatterns

This step involves the detection of ten patterns and antipatterns described in Section III. To detect these patterns and antipatterns, we follow similar detection methods in the literature [10], [11]. We also aim to improve several detection heuristics in terms of their detection performance, as discussed in Section V.

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, Figure 2 depicts the detection heuristic for ✗*Non-pertinent Documentation* antipattern. We obtain domain-specific knowledge from the documentation of each endpoint (lines 2–3) to construct a topic model [24] 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 [25] measurements.

Figure 3 shows the detection heuristics for ✗*Contextless Resource Names* antipatterns. As described in Section III, nodes in an endpoint should be semantically related [2]. 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. If all the nodes of the endpoint fall into one topic of the LDA topic model, 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. If the average similarity score of nodes is below the threshold, then 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.

Patterns and Antipatterns Detection: For detecting ten patterns and antipatterns, we utilized detection heuristics from the literature with minor modifications and improvements [10], [11]. We developed a Python-based detection tool implementing detection algorithms for ten detection heuristics. We relied on heuristics to detect antipatterns that only require syntactical analysis. Conversely, for antipatterns that require semantic analysis, we employed heuristics and various natural language processing (NLP) tools and techniques, e.g., NLTK (www.nltk.org/) and spaCy (https://spacy.io/usage/spacy-101). To detect ✗*Non-descriptive Endpoint*, ✗*Inconsistent Documentation*, ✗*Pluralized Nodes*, and ✗*CRUDy Endpoints*,

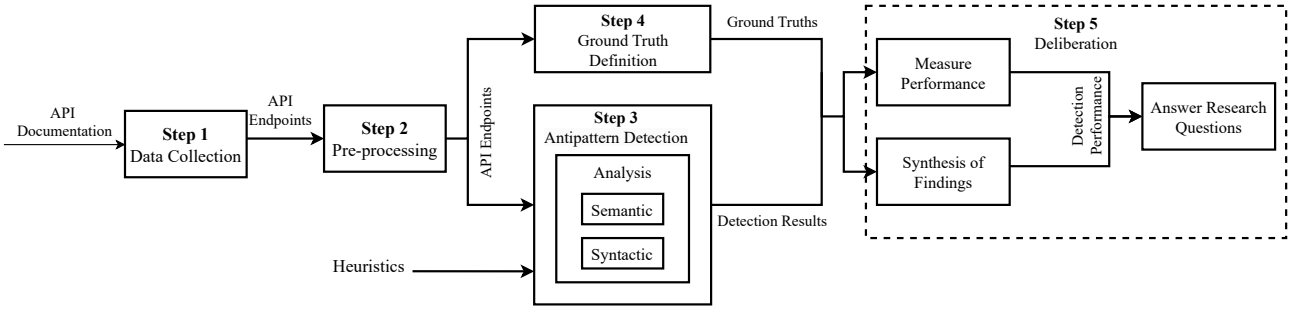

Fig. 1: Research Methodology for Assessing Linguistic Design Quality of APIs.

1: NON-PERTINENT DOCUMENTATION(*Request-Endpoint*, *Documentation*)
2: *Documentation* ← REMOVE-STOP-WORDS(*Documentation*)
3: *Tokens* ← LEMMATIZE-TOKENIZE(*Documentation*)
4: *TopicModel* ← EXTRACT-TOPICS(*Tokens*)
5: *EndpointNodes* ← EXTRACT-ENDPOINT-NODES(*Request-Endpoint*)
6: *Similarity-Value* ← COSINE-SIMILARITY-SCORE(*EndpointNodes*, *TopicModel*)
7: **if** *Similarity-Value* < *threshold*
8: **return** *'Non-Pertinent Documentation' antipattern*
9: **end if**
10: **return** *'Pertinent Documentation' pattern*

Fig. 2: *'Non-Pertinent Documentation'* detection heuristic.

1: CONTEXTLESS-RESOURCE(*Request-Endpoint, Documentation*)
2: *TopicsModel* ← EXTRACT-TOPICS(*Documentation*)
3: *EndpointNodes* ← EXTRACT-ENDPOINT-NODES(*Request-Endpoint*)
4: *Avg-Similarity-Value* ← COSINE-SIMILARITY-SCORE(*EndpointNodes, TopicsModel*)
5: **if** *Avg-Similarity-Value* < *threshold*
6: **return** *'Contextless Resource Names' antipattern*
7: **end if**
8: **return** *'Contextualized Resource Names' pattern*

Fig. 3: *'Contextless Resource Names'* detection heuristic.

we used standard pre-processing, heuristics, and regular expressions. Standard pre-processing involves removing stop words, tokenization, lowercasing, stemming, lemmatization, part-of-speech (POS) tagging, removing punctuation, and word embedding. To detect ✗*Non-hierarchical Nodes* antipattern, we utilized WordNet [26], and to detect ✗*Contextless Resource Names* and ✗*Non-pertinent Documentation*, we employed LDA topic model [24] and cosine semantic similarity metric [25].

LDA Topic Model and Cosine Semantic Similarity: We used Gensim (https://pypi.org/project/gensim/), a widely used Python library for building topic models [24]. For ✗*Contextless Resource Names* and ✗*Non-pertinent Documentation* detection, we built topic models of k topics from the documentation of the API endpoints. The topic models were examined to extract the key concepts within the API documentation. This approach allowed us to discover hidden relationships among the endpoint nodes and their documentation. The degree of similarity between two vectors in an inner product space is measured by cosine similarity [25]. It determines whether two vectors are pointing in approximately the same direction by computing the cosine of the angle between them. This measure is widely used in text analysis and natural language processing to evaluate document similarity.

D. Step 4: Ground Truth Definition

The definition of ground truth involves utilizing a random sampling approach to select 91 endpoints from a pool of 1,655 endpoints from 21 REST and 12 GraphQL APIs. This step aims to measure the performance of our implemented detection heuristics. The population comprises 16,550 endpoint instances (1,655 endpoints × 10 antipatterns). We chose a sample size of 910 queries (91 endpoints × 10 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. For each individual, we provided the HTTP method, endpoint, description, and parameters (if available). We used majority voting to select whether an endpoint has a specific antipattern.

E. Step 5: 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 RQ1, we compile a detailed detection results table and use a mosaic plot for REST and GraphQL APIs. For RQ2, we used various performance metrics such as accuracy, precision, recall, and F1-score. Finally, to answer RQ3 and RQ4, we visualize the results through the detection result table and stacked column chart, illustrating the proportion of endpoints identified as antipatterns across the two API categories for distributed systems and microservices.

V. RESULTS

This section presents our detection results and answers our research questions.

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

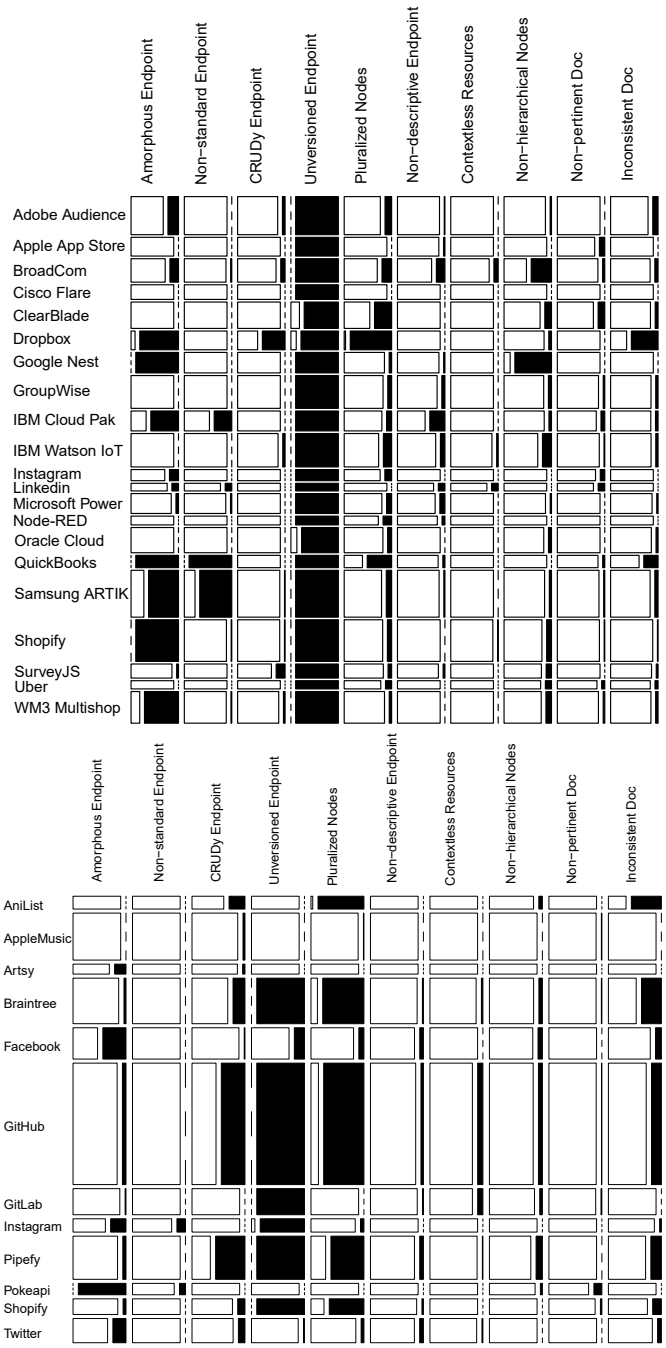

Fig. 4: Detection of Patterns and Antipatterns in REST APIs (top) and GraphQL APIs (bottom). The black portion represents antipatterns and the white portion represents patterns.

A. Overview of the Detection Results

Figure 4 depicts the mosaic plots of the detection results for ten patterns and antipatterns on 33 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 Figure 4 shows, the most prevalent antipattern in APIs is ✗*Unversioned Endpoint*, while other common antipatterns are ✗*Amorphous Endpoint*, ✗*Pluralized Nodes*, and ✗*CRUDy Endpoint*.

The detailed detection results for ten patterns and antipatterns are shown in Table II. Each column shows the detection instances for all ten 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 98% of the endpoints contain ✗*Unversioned Endpoint* antipatterns. In contrast, 99% of the endpoints follow ✓*Contextualized Resource Names* pattern. In GraphQL APIs, 68% of the endpoints follow ✗*Unversioned Endpoint* antipatterns, and 99% follow ✓*Standard Endpoint* design.

B. Poor Linguistic Design in APIs (RQ1)

RQ1 investigates the presence of poor linguistic design quality in APIs. This study only considers REST and GraphQL APIs of distributed systems and microservices. Thus, we present and analyze the poor linguistic design in APIs from the perspective of REST and GraphQL. From Table II, the most prevalent antipattern in REST APIs is ✗*Unversioned Endpoint*, constituting 98% of the detected antipatterns. Moreover, REST APIs generally follow good API design principles for linguistic patterns. For example, 88% of the endpoints follow ✓*Standard Endpoint* patterns, 95% of the endpoints are ✓*Verbless Endpoint*, and 92% of the endpoints have ✓*Consistent Documentation*. This suggests that REST API developers tend to follow good API design practices.

In GraphQL APIs, the most prevalent antipatterns include ✗*Unversioned Endpoint* (68% of the endpoints) and ✗*Pluralized Nodes* (50%). For other linguistic patterns, GraphQL follows similar design practices to REST APIs. For example, GraphQL APIs commonly follow linguistic patterns like ✓*Tidy Endpoint* (86% of the endpoints), ✓*Descriptive Endpoint* (96%), ✓*Contextualized Resource Names* (95%), ✓*Hierarchical Nodes*(94%), and ✓*Pertinent Documentation*(99%). Thus, GraphQL API developers also follow good API design practices, except for ✗*Unversioned Endpoint* and ✗*Pluralized Nodes* antipatterns.

From Figure 4, in REST APIs, the second most common antipattern is ✗*Amorphous Endpoint*. Among the other REST APIs, the majority of APIs also follow the ✓*Verbless Endpoint* pattern, except for Dropbox and SurveyJS. Similarly, most APIs adopt the ✗*Pluralized Nodes* antipattern, excluding the Apple App Store and Cisco Flare APIs. ✗*Inconsistent Documentation* antipattern is common in all APIs, except for Dropbox and QuickBooks. For GraphQL APIs, nearly all APIs exhibit ✗*Amorphous Endpoint*, except for AniList, Apple Music, and GitLab. We also found that the presence of ✗*CRUDy Endpoint* is widespread across most GraphQL APIs, except for Apple Music, Artsy, GitLab, and PokéAPI. Furthermore, developers do not always properly document their APIs, i.e., most GraphQL APIs demonstrate ✗*Inconsistent Documentation* antipattern.

TABLE II: Detection Results on 1,655 Endpoints from 33 REST and GraphQL APIs for ten Patterns and Antipatterns.

API Name	Amorphous	Tidy	Non-Standard	Standard	CRUDy	Verbless	Unversioned	Versioned	Pluralized	Singular	Non-Descriptive	Descriptive	Contextless	Contextualized	Non-Hierarchical	Hierarchical	Pertinent	Pertinent	Inconsistent	Consistent
REST																				
Adobe Audience	16	49	0	65	4	61	65	0	10	55	1	64	0	65	2	63	0	65	8	57
Apple App Store	0	32	0	32	0	32	32	0	0	32	1	31	0	32	0	32	4	28	0	32
BroadCom	8	32	1	39	4	36	40	0	9	31	8	32	4	36	19	21	2	38	3	37
Cisco Flare	0	25	0	25	0	25	25	0	0	25	0	25	0	25	0	25	0	25	0	25
ClearBlade	0	45	0	45	0	45	36	9	18	27	0	45	0	45	7	38	7	38	3	42
Dropbox	29	3	0	32	17	15	28	4	31	1	0	32	0	32	2	30	0	32	20	12
Google Nest	35	0	0	35	0	35	35	0	2	33	1	34	0	35	30	5	0	35	0	35
GroupWise	0	56	0	56	0	56	56	0	5	51	4	52	0	56	4	52	2	54	3	53
IBM Cloud Pak	22	12	14	20	0	34	34	0	4	30	12	22	0	34	2	32	3	31	2	32
IBM Watson IoT	0	57	0	57	3	54	57	0	11	46	6	51	2	55	12	45	1	56	3	54
Instagram	4	15	0	19	0	19	19	0	3	16	0	19	0	19	0	19	2	17	0	19
Linkedin	2	11	2	11	0	13	13	0	0	13	2	11	2	11	0	13	2	11	0	13
Microsoft Power	2	32	1	33	0	34	34	0	4	30	4	30	0	34	1	33	0	34	1	33
Node-RED	0	15	0	15	0	15	15	0	3	12	1	14	0	15	0	15	0	15	0	15
Oracle Cloud	0	43	0	43	0	43	37	6	4	39	0	43	0	43	3	40	0	43	2	41
QuickBooks	21	0	21	0	0	21	21	0	12	9	1	20	0	21	0	21	1	20	7	14
Samsung ARTIK	56	24	60	20	1	79	80	0	10	70	2	78	0	80	4	76	2	78	6	74
Shopify	71	0	1	70	1	70	71	0	7	64	0	71	1	70	8	63	1	70	1	70
SurveyJS	1	23	0	24	5	19	24	0	2	22	1	23	0	24	3	21	0	24	1	23
Uber	0	14	0	14	0	14	14	0	2	12	0	14	0	14	2	12	1	13	1	13
WM3 Multishop	43	11	1	53	2	52	54	0	3	51	0	54	0	54	7	47	2	52	3	51
GraphQL																				
AniList	0	27	0	27	9	18	0	27	26	1	0	27	0	27	2	25	0	27	17	10
Apple Music	0	99	0	99	4	95	0	99	1	98	0	99	0	99	0	99	1	98	0	99
Artsy	5	16	0	21	1	20	0	21	0	21	0	21	0	21	0	21	0	21	0	21
Braintree	4	92	0	96	24	72	96	0	83	13	2	94	3	93	7	89	1	95	40	56
Facebook	32	34	0	66	1	65	14	52	7	59	5	61	0	66	6	60	0	66	8	58
GitHub	19	237	0	256	127	129	256	0	216	40	13	243	28	228	21	235	2	254	53	203
GitLab	1	54	0	55	0	55	55	0	0	55	1	54	6	49	3	52	1	54	0	55
Instagram	9	19	5	23	0	28	26	2	2	26	0	28	0	28	0	28	0	28	1	27
Pipefy	6	85	0	91	56	35	91	0	63	28	7	84	1	90	12	79	0	91	20	71
PokéAPI	24	0	3	21	0	24	0	24	0	24	0	24	0	24	1	23	4	20	0	24
Shopify	2	31	0	33	5	28	33	0	24	9	1	32	0	33	0	33	1	32	6	27
Twitter	14	37	0	51	7	44	1	50	3	48	4	47	1	50	0	51	0	51	4	47

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

C. Accuracy of Detection Algorithms (RQ2)

RQ2 aims to investigate the detection accuracy of our detection algorithms. The detection accuracy for each of the ten patterns and antipatterns is shown in Table III. On a set of 91 endpoints (i.e., 91 $\times$ 10 instances of antipattern), our detection algorithms achieved an average detection accuracy of 93.08%, precision of 79.9%, and recall of 86.59% (thus, average F1-score of 85.98%). Validation results suggest that our detection algorithms outperform state-of-the-art detection methods [10], [11].

The state-of-the-art approaches SARA [10] achieved an average F1-score of 80.9% and DOLAR [9] achieved 79.5%. SARAv2 [11] (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., 91). The detection algorithms implemented in this study achieved a better performance than state-of-the-art methods in terms of both F1-score and accuracy. Our study outperformed DOLAR and SARA in terms of F1-score by a margin of 6.5% and 5%. Our detection algorithms also outperformed SARAv2 by a margin 21.98% 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* [27] metric to capture the similarity between words, improving the overall semantic analysis.

However, the performance metrics are also influenced by how developers, in this case, three professionals, understand and interpret a word based on their experience and knowledge. From Table III, we can observe that ✗*Contextless Resource Names*, ✗*Non-pertinent Documentation*, ✗*Non-descriptive Endpoint*, ✗*Non-hierarchical Nodes*, and ✗*Pluralized Nodes* have significantly low detection performance compared to other 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" was detected as ✗*Non-hierarchical Nodes* antipattern, however, based on the majority voting, the manual validation (ground truth definition) identified the endpoint as

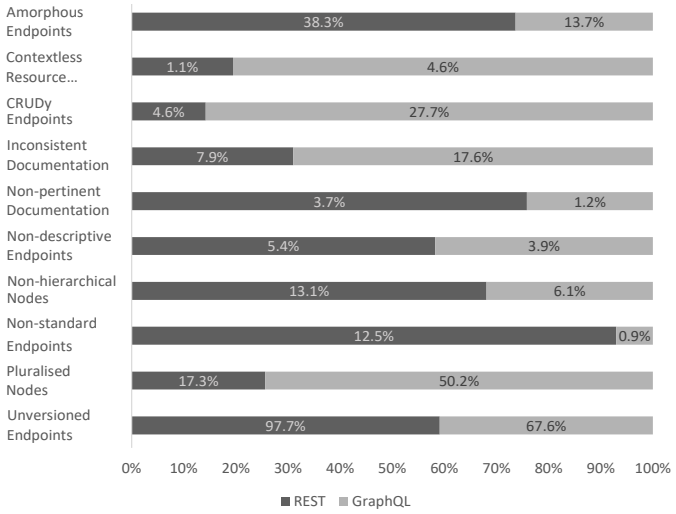

Fig. 5: Prevalence of antipatterns in REST and GraphQL APIs

✓*Hierarchical Nodes* pattern. Consequently, ✗*Non-hierarchical Nodes* antipattern exhibits a low precision, recall, and F1-score due to such interpretation conflicts among the professionals. We provide all our study resources in this GitHub repository[2].

RQ2 Summary: Our detection algorithms achieved an average accuracy of 93.08%, precision of 79.9%, recall of 86.59%, and F1-score of 85.98%. Compared to the state-of-the-art methods, our detection algorithms yield better detection performance.

D. API Category Prone to Poor Linguistic Design (RQ3)

RQ3 aims to identify the API category that is more susceptible to poor linguistic design. Figure 5 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 ten antipatterns. We observe from Figure 5 that REST APIs contain more antipatterns than GraphQL APIs. More specifically, endpoints from GraphQL APIs had 1,638 instances of antipatterns (out of 8,470 = 847 endpoints × 10 antipatterns). In comparison, endpoints from REST APIs had 1,631 instances of antipatterns (out of 8,090 = 809 endpoints × 10 antipatterns), which suggests overall 17% of the REST endpoints contain antipatterns, in contrast to 16% for GraphQL endpoints. Thus, REST APIs have a slightly higher proportion of antipattern instances than GraphQL APIs.

RQ3 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.

[2]https://github.com/krishnodey/CASCON-Supplementary-Material

E. Most Common Linguistic Patterns and Antipatterns (RQ4)

We aim to find which linguistic patterns and antipatterns are more common in APIs of distributed systems and microservices through RQ4. From Table II and Figure 5, the most common antipatterns are ✗*Unversioned Endpoint*, ✗*Amorphous Endpoint*, and ✗*Pluralized Endpoint*. In contrast, common good design practices among the APIs are ✓*Descriptive Endpoint*, ✓*Contextualized Resource Names*, and ✓*Pertinent Documentation*. Linguistic antipatterns that are rare in REST APIs are ✗*Non-descriptive Endpoints* (5% of endpoints), ✗*Contextless Resource Names* (1% of endpoints), and ✗*Non-pertinent Documentation* (4% of endpoints). On the other hand, for GraphQL APIs, ✗*Contextless Resource Names* (4% of endpoints), ✗*Non-standard Endpoints* (1% of endpoints), and ✗*Pertinent Documentation* (5% of endpoints) antipatterns are rare. Moreover, in REST and GraphQL APIs, ✓*Standard Endpoints* and ✓*Hierarchical Nodes* are common good practices.

RQ4 Summary: Most commonly occurring antipatterns are ✗*Unversioned Endpoint*, ✗*Amorphous Endpoint*, and ✗*Pluralized Nodes*, 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.

VI. Discussions

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. We also discuss the implications of our study and identify threats that may affect the validity of our results.

In total, 29 out of 33 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

TABLE III: 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	33	58	30	3	58	0	96.7%	90.91%	100%	93.72%
Contextless Resource Names	3	88	2	1	82	6	92.31%	66.67%	25%	77.42%
CRUDy Endpoint	17	74	15	2	74	0	97.8%	88.24%	100%	92.77%
Inconsistent Documentation	14	77	11	3	69	8	87.91%	78.57%	57.89%	82.98%
Non-descriptive Endpoint	3	88	2	1	86	2	96.7%	66.67%	50%	78.92%
Non-hierarchical Nodes	12	79	6	6	77	2	91.21%	50%	75%	64.59%
Non-standard Endpoint	8	83	7	1	83	0	98.9%	87.5%	100%	92.85%
Non-pertinent Documentation	4	87	3	1	87	0	98.9%	75%	100%	85.31%
Pluralized Nodes	27	64	6	21	58	6	70.33%	22.22%	50%	33.77%
Unversioned Documentation	73	18	73	0	18	0	100%	100%	100%	100%
Average							**93.08%**	**79.9%**	**86.59%**	**85.98%**

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 not misleading.

✗*Amorphous Endpoint* antipattern is one the most prevalent antipatterns, with 25 out of 33 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.

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

A. Implications for Developers

Application developers usually review 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 understanding and use. This study 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 our study 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. Our other observations include: ✗*Amorphous Endpoint*, ✗*Pluralized Nodes*, and ✗*CRUDy Endpoint*, i.e., API developers commonly include amorphous design, pluralized nodes, and CRUDy verbs in endpoint design. In summary, API developers could use our findings to improve the overall design quality of their APIs.

B. Threats to Validity

Our research mainly focuses on linguistic patterns and antipatterns in APIs of distributed systems and microservices. We performed experiments using a dataset that included 1,655 endpoints from 33 REST and GraphQL APIs to minimize threats to the external validity of our findings. However, to confirm the results further, we need to consider more APIs and endpoints. Our detection algorithms can identify linguistic antipatterns with an average accuracy of 93%. 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 91 endpoints out of 1,655 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 91 endpoints for ten antipatterns, i.e., $91 \times 10 = 910$ questions.

Furthermore, the 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 endpoints that were well-structured and documented. We performed a thorough analysis of linguistic patterns and antipatterns to define the detection heuristics and further 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. To minimize further threats to validity and increase reliability and reproducibility, we published all our study resources.

VII. Conclusion and Future Work

This study 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 patterns are easy to understand and adopt, while those with antipatterns hinder understanding and adoption. We analyzed 1,655 endpoints from 33 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). Our detection algorithms achieved an average accuracy of 93.08%, a precision of 79.9%, a recall of 86.59%, and an F1-score of 85.98% (RQ2). Moreover, we observed that both REST and GraphQL APIs are prone to linguistic antipatterns, with REST having slightly more antipatterns (RQ3). Finally, ✗*Unversioned Endpoint*, ✗*Amorphous Endpoint*, and ✗*Pluralized Nodes* are the most commonly occurring antipatterns in APIs (RQ4).

As part of future work, further investigation is required to refine our detection algorithms to improve the detection performance. We also plan to analyze more APIs and endpoints of other APIs to investigate their linguistic design quality. Furthermore, we also aim to define new patterns and antipatterns and develop heuristics for their detection. Definition and the detection of heuristics of new patterns and antipatterns would increase the quality of analysis of the linguistic design of API.

References

[1] Sam Newman. *Building microservices*. O'Reilly Media, Inc., 2021.

[2] Mark Masse. *REST API design rulebook: designing consistent RESTful web service interfaces*. O'Reilly Media, Inc., 2011.

[3] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[4] Graphql.org. Introduction to graphql, 2021.

[5] Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. Are rest apis for cloud computing well-designed? an exploratory study. In *Service-Oriented Computing: 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings 14*, pages 157–170. Springer, 2016.

[6] Michael Hausenblas. *On Entities in the Web of Data*, pages 425–440. Springer New York, New York, NY, 2011.

[7] Allison Parrish. Social network apis: A revised lexical analysis.

[8] Carlos Rodríguez, Marcos Baez Gonzalez, 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*, page 21–39. Lecture Notes in Computer Science. Springer International Publishing, 2016.

[9] Francis Palma, Javier Gonzalez-Huerta, Naouel Moha, Yann-Gaël Guéhéneuc, and Guy Tremblay. Are restful apis well-designed? detection of their linguistic (anti) patterns. In *Service-Oriented Computing: 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings 13*, pages 171–187. Springer, 2015.

[10] 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(02):1742001, 2017.

[11] 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*, 191:111369, 2022.

[12] Francis Palma, Tobias Olsson, Anna Wingkvist, Fredrik Ahlgren, and Daniel Toll. Investigating the linguistic design quality of public, partner, and private rest apis. In *2022 IEEE International Conference on Services Computing (SCC)*, pages 20–30. IEEE, 2022.

[13] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, 14(4):957–970, 2018.

[14] Florian Haupt, Frank Leymann, Anton Scherer, and Karolina Vukojevic-Haupt. A framework for the structural analysis of rest apis. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 55–58. IEEE, 2017.

[15] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology*, 33(1):1–41, 2023.

[16] Natalie Kiesler and Daniel Schiffner. What is a good api? a survey on the use and design of application programming interfaces. In *International Conference on Internet of Everything*, pages 45–55. Springer, 2024.

[17] Shang-Pin Ma, Ming-Jen Hsu, Hsiao-Jung Chen, and Chuan-Jie Lin. Restful api analysis, recommendation, and client code retrieval. *Electronics*, 12(5):1252, 2023.

[18] Erik Wittern, Alan Cha, James C Davis, Guillaume Baudart, and Louis Mandel. An empirical study of graphql schemas. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*, pages 3–19. Springer, 2019.

[19] Piotr Roksela, Marek Konieczny, and Slawomir Zielinski. Evaluating execution strategies of graphql queries. In *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, pages 640–644. IEEE, 2020.

[20] Budi Santosa, Awang Hendrianto Pratomo, Riski Midi Wardana, Shoffan Saifullah, and Novrido Charibaldi. Performance optimization of graphql api through advanced object deduplication techniques: A comprehensive study. *Journal of Computing Science and Engineering*, 17(4):195–206, 2023.

[21] Antonio Quiña-Mera, Pablo Fernandez, José María García, and Antonio Ruiz-Cortés. Graphql: A systematic mapping study. *ACM Computing Surveys*, 55(10):1–35, 2023.

[22] Antonio Quina-Mera, Pablo Fernández-Montes, José María García, Edwin Bastidas, and Antonio Ruiz-Cortés. Quality in use evaluation of a graphql implementation. In *XV Multidisciplinary International Congress on Science and Technology*, pages 15–27. Springer, 2021.

[23] Goran Mauša, Tihana Galinac-Grbac, and Bojana Dalbelo-Bašić. A systematic data collection procedure for software defect prediction. *Computer Science and Information Systems*, 13(1):173–197, 2016.

[24] Mark Steyvers and Tom Griffiths. Probabilistic topic models. *Handbook of latent semantic analysis*, 427(7):424–440, 2007.

[25] Faisal Rahutomo, Teruaki Kitasuka, and Masayoshi Aritsugi. Semantic cosine similarity. In *The 7th international student conference on advanced science and technology ICAST*, volume 4, page 1, 2012.

[26] Christiane Fellbaum. *WordNet: An electronic lexical database*. MIT press, 1998.

[27] Peter Kolb. Disco: A multilingual database of distributionally similar words. *Proceedings of KONVENS-2008, Berlin*, 156, 2008.