Investigating the Linguistic Design Quality of Public, Partner, and Private REST APIs

Francis Palma, Tobias Olsson, Anna Wingkvist, Fredrik Ahlgren, and Daniel Toll
Department of Computer Science and Media Technology
Linnaeus University
Kalmar, Sweden
{francis.palma, tobias.olsson, anna.wingkvist, fredrik.ahlgren, daniel.toll}@lnu.se

2022 IEEE International Conference on Services Computing (SCC) | 978-1-6654-8146-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SCC55611.2022.00017

***Abstract*—Application Programming Interfaces (APIs) define how Web services, middle-wares, frameworks, and libraries communicate with their clients. An API that conforms to REpresentational State Transfer (REST) design principles is known as REST API. At present, it is an industry-standard for interaction among Web services. There exist mainly three categories of APIs: public, partner, and private. Public APIs are designed for external consumers, whereas partner APIs are designed aiming at organizational partners. In contrast, private APIs are designed solely for internal use. The API quality matters regardless of their category and intended consumers. To assess the (linguistic) design of APIs, researchers defined *linguistic patterns* (i.e., best API design practices) and *linguistic antipatterns* (i.e., poor API design practices.) APIs that follow linguistic patterns are easy to understand, use, and maintain. In this study, we analyze and compare the design quality of public, partner, and private APIs. More specifically, we made a large survey by analyzing and performing the detection of nine linguistic patterns and their corresponding antipatterns on more than 2,500 end-points from 37 APIs. Our results suggest that (1) public, partner, and private APIs lack quality linguistic design, (2) among the three API categories, private APIs lack linguistic design the most, and (3) end-points are amorphous, contextless, and non-descriptive in partner APIs. End-points have contextless design and poor documentation regardless of the API categories.**

***Index Terms*—REST APIs, Linguistic Design, Public, Partner, Private, Patterns, Antipatterns**

I. Introduction

APIs (Application Programming Interfaces) enable communication among the Web services, middle-wares, frameworks, and libraries with their clients. An API that conforms to REST (REpresentational State Transfer) principles are commonly known as REST API [1]. REST at present is an industry-standard protocol for the interaction among the Web services and their clients. REST relies on HTTP (Hypertext Transfer Protocol) requests and responses for its operation. A REST request contains an end-point comprising a domain, port, path, and/or query string together with an HTTP method.

APIs can be categorized depending on their intended target audience [2]. We find three main categories of audiences, namely public, partner, and private . A public API is designed to be accessible by anyone interested in the API without a close relationship. In the context of public APIs, the primary goal is to make consumption easy such that numerous users (or clients) are attracted. In contrast, a private API is primarily intended for internal use in an organization to share back-end data and application functionalities among users [2]. Thus, private APIs are not disclosed to external users and usually require authentication to access them. Likewise, partner APIs are available only to the selected/authorized external API users and facilitate business-to-business activities. Thus, our initial hypothesis is that public APIs are of higher design quality than the partner and private APIs, since they aim to attract external users.

To facilitate use of REST APIs, their linguistic design quality is essential. The lack of linguistic design quality is synonymous to linguistic antipatterns [3]. More specifically, Arnaoudova et al. [3] defined *linguistic antipatterns* as: "*recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding*". An API that conforms to the REST design principles would arguably be easier to use and understand, would create less frustration for application developers, and more widespread use of API [4]. To assess the linguistic design quality of REST APIs, researchers defined *linguistic patterns* (i.e., best practices of REST API design) and *linguistic antipatterns* (i.e., poor REST API design practices to avoid). REST APIs that follow linguistic patterns are easier to understand and use. In contrast, linguistic antipatterns hinder the understanding and use of REST APIs.

High linguistic design quality would, logically, be more important for public and partner APIs compared to private APIs as public and partner APIs are intended for use outside of the developing organization. Private API quality would impact the developing organization itself; however, a lack of quality could be overcome by, for example, direct communication between developers, or there may be standard internal ways of designing an API that are not captured in the REST design principles. It can also be the case that some principles of REST design are prioritized differently by practitioners depending on intended audience. Kotstein and Bogner [5] suggest that practitioners do not regard all REST design principles as equally important. It could also be that design quality costs development time, which is not well motivated for internal APIs.

Researchers assessed and studied the quality of APIs, in particular for public APIs [6], [7], [8], [5], [9]. However, to the best of our knowledge, no study analyzed linguistic design of private and partner categories of APIs. Moreover, it is un-

2474-2473/22/$31.00 ©2022 IEEE
DOI 10.1109/SCC55611.2022.00017

known whether API providers for the three categories of APIs are equally concerned with the design of their APIs. More specifically, in this study, we want to investigate and compare the linguistic design of public, partner, and private APIs. We aim to discover whether a certain API category is more prone to linguistic antipatterns and which linguistic antipatterns are more common across APIs categories. Findings from our study will inform API developers of the area they need to focus on to improve the design quality of their APIs.

Our study reflects on general landscape of API design showing the current state of design of APIs. As we perform the analyses for three categories of APIs, a comparative view shows which category of APIs relatively follows more good design practices than other API categories. More specifically, we aim to answer the following three research questions:

- **RQ$_1$:** To what extent do the public, partner, and private APIs suffer from poor linguistic design, i.e., linguistic antipatterns?
  RQ$_1$ aims to investigate whether public, partner, and private APIs lack linguistic quality.
- **RQ$_2$:** Among the public, partner, and private APIs, which category of APIs is more prone to poor linguistic design?
  RQ$_2$ aims to find whether one category of API is more prone to poor linguistic quality.
- **RQ$_3$:** Among the nine linguistic patterns and antipatterns, which of them occur most in the three categories of APIs?
  RQ$_3$ aims to identify linguistic patterns or antipatterns more common in public, partner, and private APIs.

Main contributions of this study are (1) empirical evidence that poor linguistic design is prevalent among public, partner, and private APIs; (2) an analysis of more than 2,500 end-points in the form of the detection of nine linguistic patterns and their corresponding antipatterns from 37 public, partner, and private APIs; (3) definition of a new linguistic pattern and its corresponding antipattern; and (4) identifying which category of APIs are more prone to linguistic antipatterns and which antipatterns are more common than others. We found private APIs more prone to linguistic antipatterns than public and partner APIs, and the design between public and partner APIs differs but is statistically insignificant.

In the rest of the paper: Section II provides a brief introduction to the linguistic patterns and antipatterns detected. Section III shows the design of our study while Section IV presents the detection results and our findings. Section V makes a high-level discussion and outlines the threats to the validity of our results. Section VI highlights the work related to the assessment of API design. Finally, Section VII concludes the research and provides future work.

II. Linguistic Patterns and Antipatterns in APIs

This section briefly introduces nine linguistic patterns and antipatterns. The linguistic pattern ✔*Self-descriptive end-point* and its corresponding antipattern ✖*Non-descriptive end-point* is newly defined in this study.

A. Tidy vs. Amorphous End-point

End-points in REST should be tidy and easy to read. A ✔*Tidy End-point* has an appropriate lower-case resource naming, no extensions, underscores, or trailing slashes. ✖*Amorphous End-point* occurs when an end-point contains symbols or capital letters that make them difficult to read and use. An end-point is amorphous if it contains: (1) upper-case letter (except for Camel Cases [10]), (2) file extensions, (3) underscores, and, (4) a final trailing-slash [4], [11].
Example: The end-point /NEW_Customer/__image01.tiff/ is an ✖*Amorphous End-point* while the end-point /customers/1234 is a ✔*Tidy End-point*.

B. Contextualized vs. Contextless Resource Names

End-points should be *contextual*, *i.e.*, nodes in end-points should be semantically-related. ✖*Contextless Resource Names* antipattern appears when end-points are composed of nodes that do not belong to the same semantic context [12]. In contrast, if the nodes in an end-point belong to same semantic context this is known as ✔*Contextual Resource Names* pattern.
Example: The end-point /newspapers/planet/players?id=123 is ✖*Contextless Resource* while the end-point /soccer/team/players?id=123 is ✔*Contextual Resource*.

C. Verbless vs. CRUDy End-point

Appropriate HTTP methods, *e.g.*, GET, POST, PUT, or DELETE, should be used to perform an action. The use of CRUDy terms (*e.g.*, create, read, update, delete, or their synonyms) as part of the end-point design is highly discouraged [4], [12]. This practice is known as ✖*CRUDy End-point*. In contrast, when design an end-point without any CRUDy term and use an appropriate HTTP method in pair with the end-point, this is known as ✔*Verbless End-point* pattern [6].
Example: The end-point /update/players/age?id=123 is a ✖*CRUDy End-point* while the end-point /players/age?id=123 is a ✔*Verbless End-point*.

D. Consistent vs. Inconsistent Documentation

The ✖*Inconsistent Documentation* antipattern occurs when an end-point (together with the HTTP method) is in contradiction with its documentation. When the end-point and underlying HTTP method aligns with its documentation, this is known as ✔*Consistent Documentation* pattern [6].
Example: The POST method with the end-point /bulk/devices/remove is in contradiction with its documentation '*Delete multiple devices. Delete multiple devices, each request can contain a maximum of 512kB*', thus, is an ✖*Inconsistent Documentation*. Whereas, the end-point /bulk/devices/remove with the documentation '*Remove multiple devices. Remove multiple devices, each request can contain a maximum of 512kB*', would be identified as ✔*Consistent Documentation*.

E. Standard vs. Non-standard End-point

An End-point design should not include nodes/resources with non-standard identification, which hinders the reusability and understandability of APIs. The ✖*Non-standard End-point*

antipattern occurs when (1) characters like é, å, ö, etc. are present in URIs, (2) blank spaces are found in end-points, (3) double hyphens are used in end-points, and (4) unknown characters (e.g., !, @, #, $, %, ^, &, *, etc.) are present in end-points. Instead, an end-point following ✔*Standard End-point* design (1) does not include non-standard characters like é, å, ö, etc. and (2) replaces blank spaces, unknown characters, and double hyphens with a single hyphen.

Example: The end-point /museum/louvre/réception/ is an example of ✖*Non-standard URI Design*. While, the end-point /museum/louvre/reception/ represents ✔*Standard URI Design*.

F. Pertinent vs. Non-pertinent Documentation

The ✖*Non-pertinent Documentation* antipattern occurs when the documentation of an end-point is not cohesive to the design of the end-point itself. In contrast, a well-documented end-point properly and clearly describe its purpose using semantically related terms can be considered as ✔*Pertinent Documentation* pattern [13].

Example: The end-point and documentation pair from Twitter: api.twitter.com/1.1/favorites/list – *'Returns the 20 most recent Tweets liked by the authenticating or specified user'* is considered as a ✖*Non-pertinent Documentation*. In contrast, this end-point and documentation pair from Instagram: instagram.com/media/media-id/comments – '*Gets a list of recent comments on a media object. The public content permission scope is required to get comments for a media that does not belong to the owner of the access token.*' is considered as a ✔*Pertinent Documentation*.

G. Hierarchical vs. Non-hierarchical Nodes

Nodes in an end-point should be hierarchically related to its neighbor nodes. ✖*Non-hierarchical Nodes* is an antipattern that appears when at least one node in an end-point is not hierarchically related to its neighbor nodes [12]. When all the nodes in an end-point are in a hierarchical relationship, this is known as ✔*Hierarchical Nodes* pattern.

Example: The end-point /professors/faculty/university is an example of ✖*Non-hierarchical Nodes* while the end-point /university/faculty/professors is an example of ✔*Hierarchical Nodes*.

H. Singularized vs. Pluralized Nodes

End-points should use singular/plural nouns consistently for resources naming across the API. When clients send PUT or DELETE requests, the last node of the request end-point should be singular. In contrast, for POST requests, the last node should be plural. Therefore, the ✖*Pluralized Nodes* antipattern appears when plural names are used for PUT/DELETE requests or singular names are used for POST requests, otherwise it can be considered as ✔*Singularized Nodes* pattern [6].

Example: The POST method with the end-point /team/player is a ✖*Pluralized Nodes* while the POST method with the end-point /team/players is a ✔*Singularized Nodes*.

I. Self-descriptive vs. Non-descriptive End-point

The end-points in REST APIs design should be as human-friendly as possible, *i.e.*, with the end-users in mind, and the search engine optimization and ease of development should come second. An end-point should be short and to the point, *i.e.*, as few characters as possible while still maintaining usability. ✖*Non-descriptive End-point* antipattern occurs when the end-point design has encoded nodes, *e.g.*, simple resource names not used, which hinder its understandability. In contrast, a ✔*Self-descriptive End-point* has clear and concise resource identifiers.

Example: The end-point /auth/token/from_oauth1 is a ✖*Non-descriptive* while the end-point /account/set_profile_photo is a ✔*Self-descriptive*.

III. Study Design

This section presents the study design and the REST APIs we analyzed.

A. Data Collection and Processing

Figure 1 shows the steps followed to answer our research questions. The steps are briefly described in the following.

Step 1: Data Collection involves the collection of details for the APIs from three categories: public, partner, and private. In particular, for each API, we manually gather the end-points, underlying HTTP methods, and available documentation of these end-points. We build a dataset that contains the end-points, underlying HTTP methods, and documentation for 2,519 end-points from 37 APIs. More specifically, our dataset contains 1,443 end-points from 21 public APIs, 669 end-points from 9 partner APIs, and 407 end-points from 7 private APIs. The dataset is fed as input to Step 2.

Step 2: Detection of Linguistic Patterns and Antipatterns performs to find nine linguistic patterns and their corresponding antipatterns with the help of the SARA approach [6] and the SOFA framework [14]. The SARA approach automatically detects linguistic patterns and antipatterns in REST APIs for Web and cloud services. SARA involves three main steps: (1) definition of detection heuristics for linguistic patterns and antipatterns, (2) implementation of detection algorithms for linguistic patterns and antipatterns, and (3) automatic detection of linguistic patterns and antipatterns.

The detection of linguistic patterns and antipatterns utilizes natural language processing techniques and dictionary-based analysis to assess the linguistic design of end-points in APIs. SARA has an average precision and an average recall of more than 80% [6]. The SOFA framework provides the detection environment for patterns and antipatterns in service-based systems. At present, SOFA enables automatic detection of 51 patterns and antipatterns [14]. As an outcome of Step 2, for each API, we produce the list of end-points with information showing the presence or absence of linguistic patterns and antipatterns.

Step 3: Descriptive Analyses summarize detection results to answer our three research questions. In short, to answer

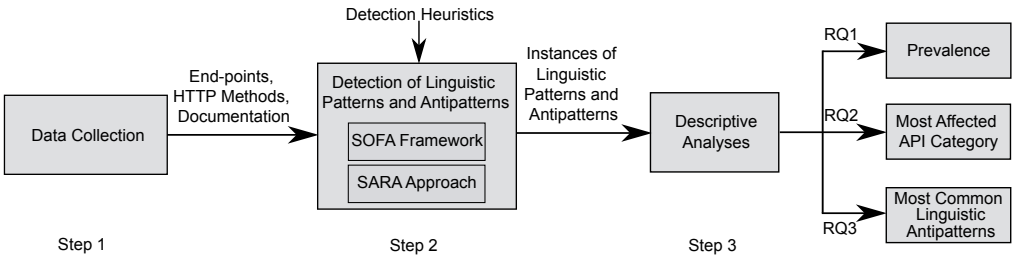

Fig. 1. Study design.

RQ1, we create an aggregated table for public, partner, and private APIs showing the total number of end-points that are detected as patterns and antipatterns. To answer RQ2, we summarize detection results using the proportion of end-points for each API. Since our dataset has varying number of end-points for the APIs, we also compute Normalized Antipatterns Rate (NAR), for each API, based on Equation 1.

$$NAR = \frac{\sum Antipattern_i}{(EndPoint \times N)} \quad (1)$$

The index *i* ranges from 1 to *N*; *N* is the number of antipatterns detected; $Antipattern_i$ is the number of instances detected for the *i*-th antipattern; and $EndPoint$ is the number of end-points analyzed for the API. A high NAR value should be interpreted as having a high occurrence of antipatterns over the endpoints of an API.

To check and further confirm the significance of the differences in NAR among the API groups, we perform a two-tailed Welch's T-test [15]. We use this test because the number of partner and private APIs is relatively low compared to public APIs and the variance is unequal between the populations of different sizes, making tests like 3-way ANOVA not suitable due to violated assumptions. We perform the tests with the *median* effect size and at the significance level of 0.05. An effect size of $d \geq 0.2$ is referred to as *small*, $d \geq 0.5$ as *medium*, and $d \geq 0.8$ as *large* [16]. To answer RQ3, we plot a stacked column chart based on the percentage of end-points detected as antipatterns for three categories of APIs.

B. APIs under Analysis

To answer our three research questions, we rely on 37 public, partner, and private APIs. We choose public APIs that are well-known and end-points have well-organized documentation. Our set of public APIs includes very well-known REST APIs for Web applications (e.g., Dropbox, Facebook, Instagram, Twitter, YouTube, and Stack Overflow) and APIs for applications related to IoT or Internet of Things (e.g., Amazon AWS Core IoT, Arduino IoT Cloud, IBM Watson IoT, Microsoft Azure IoT). In general, it is easy to find public and partner APIs. In contrast, since private APIs are developed for internal purposes, they are hard to discover. After a thorough search, we gathered seven private APIs. Table I shows the names and documentation of 37 public, partner, and private APIs and the number of end-points analyzed.

TABLE I
LIST OF 37 APIS AND THE NUMBER OF END-POINTS ANALYZED.

API Names	API Types	End-points Analyzed
Amazon AWS Core IoT	*public*	150
Arduino IoT Cloud	*public*	20
Cisco Flare	*public*	34
ClearBlade	*public*	84
Dropbox	*public*	48
Droplit.io	*public*	52
Facebook Graph API	*public*	65
Google Nest	*public*	47
IBM Watson IoT	*public*	139
Instagram	*public*	21
Linkedin	*public*	13
Losant	*public*	63
Microsoft Azure IoT	*public*	210
Microsoft Power BI	*public*	52
Node-RED	*public*	17
Samsung ARTIK Cloud	*public*	137
Sonos	*public*	49
Stack Overflow	*public*	77
The Things Network	*public*	11
Twitter	*public*	104
YouTube	*public*	50
Sub Total		**1,443**
IBM Cloud Pak System	*partner*	52
LiveAgent	*partner*	21
Microsoft Partner Center	*partner*	70
Oracle Cloud Marketplace Publisher	*partner*	77
Prolateral Core Infrastructure	*partner*	27
QuickBooks Online	*partner*	26
Shopify	*partner*	272
Uber	*partner*	15
WM3 Multishop	*partner*	109
Sub Total		**669**
Adobe Audience Manager	*private*	141
Apple App Store Connect	*private*	48
Bitholic	*private*	6
BroadCom	*private*	40
GroupWise	*private*	135
Guardian News	*private*	5
SurveyJS	*private*	32
Sub Total		**407**
Total		**2,519**

IV. CASE STUDY RESULTS

This section summarizes our findings and answers our research questions.

A. Poor Linguistic Design in APIs (RQ_1)

RQ_1 investigates whether public, partner, and private APIs lack linguistic quality and to what extent. Table II shows the detection results of nine linguistic antipatterns and corresponding patterns on 37 APIs. Columns in the table show detection results for each pair of linguistic antipattern and their corresponding pattern. Rows represent the APIs under

TABLE II
DETECTION RESULTS OF NINE PATTERNS AND CORRESPONDING ANTIPATTERNS ON 2,519 END-POINTS FROM 37 PUBLIC, PARTNER, AND PRIVATE APIS.

APIs	✖Amorphous End-point	✔Tidy End-point	✖Contextless Resources	✔Contextualized Resources	✖CRUDy End-point	✔Verbless End-point	✖Non-descriptive End-point	✔Self-descriptive End-point	✖Inconsistent Doc.	✔Consistent Doc.	✖Non-standard End-point	✔Standard End-point	✖Non-pertinent Doc.	✔Pertinent Doc.	✖Non-hierarchical Nodes	✔Hierarchical Nodes	✖Pluralized Nodes	✔Singularized Nodes
Public APIs																		
Amazon AWS Core IoT	0	150	13	137	9	141	4	146	8	142	0	150	113	37	0	150	39	111
Arduino IoT Cloud	0	20	0	20	0	20	0	20	7	13	0	20	9	11	0	20	5	15
Cisco Flare	0	34	0	34	0	34	0	34	0	34	0	34	20	14	0	34	4	30
ClearBlade	0	84	0	84	0	84	9	75	3	81	0	84	66	18	0	84	23	61
Dropbox	0	48	2	46	4	44	2	46	16	32	0	48	16	32	0	48	48	0
Droplit.io	7	45	1	51	1	51	2	50	1	51	0	52	21	31	0	52	12	40
Facebook	0	65	7	58	0	65	8	57	0	65	0	65	30	35	0	65	11	54
Google Nest	0	47	4	43	0	47	4	43	0	47	0	47	29	18	0	47	2	45
IBM Watson IoT	0	139	4	135	3	136	10	129	1	138	0	139	82	57	0	139	27	112
Instagram	0	21	2	19	0	21	1	20	0	21	0	21	19	2	0	21	4	17
Linkedin	0	13	2	11	0	13	1	12	0	13	2	11	12	1	0	13	4	9
Losant	0	63	7	56	2	61	1	62	2	61	0	63	15	48	0	63	23	40
Microsoft Azure IoT Hub	0	210	74	136	3	207	7	203	59	151	0	210	210	0	0	210	42	168
Microsoft Power BI	0	52	1	51	1	51	15	37	0	52	1	51	18	34	0	52	15	37
Node-RED	0	17	0	17	0	17	1	16	0	17	0	17	0	17	0	17	5	12
Samsung ARTIK Cloud	0	137	19	118	1	136	5	132	2	135	108	29	71	66	0	137	29	108
Sonos	0	49	2	47	2	47	0	49	0	49	0	49	27	22	0	49	35	14
Stack Overflow	0	77	9	68	6	71	0	77	5	72	0	77	9	68	0	77	30	47
The Things Network	0	11	0	11	0	11	7	4	0	11	0	11	5	6	0	11	4	7
Twitter	2	102	3	101	32	72	6	98	0	104	0	104	31	73	0	104	38	66
YouTube	0	50	1	49	1	49	1	49	1	49	0	50	15	35	0	50	16	34
Percentage (in %)	1	99	10	90	5	95	6	94	7	93	8	92	57	43	0	100	29	71
Partner APIs																		
IBM Cloud Pak System	0	52	19	33	0	52	17	35	1	51	14	38	33	19	0	52	16	36
LiveAgent	0	21	6	15	0	21	5	16	0	21	0	21	16	5	0	21	3	18
Microsoft Partner Center	0	70	14	56	0	70	20	50	1	69	0	70	42	28	0	70	18	52
Oracle Cloud Mp Publisher	0	77	10	67	0	77	0	77	1	76	0	77	49	28	0	77	14	63
Prolateral Core Infrastructure	0	27	0	27	0	27	3	24	0	27	0	27	4	23	0	27	0	27
QuickBooks Online	0	26	0	26	0	26	1	25	0	26	26	0	18	8	0	26	14	12
Shopify	271	1	272	0	0	272	271	1	3	269	1	271	272	0	0	272	71	201
Uber	0	15	2	13	0	15	0	15	0	15	0	15	3	12	0	15	2	13
WM3 Multishop	0	109	11	98	1	108	0	109	2	107	1	108	74	35	0	109	22	87
Percentage (in %)	41	59	50	50	0	100	47	53	1	99	6	94	76	24	0	100	24	76
Private APIs																		
Adobe Audience Manager	33	108	14	127	0	141	4	137	13	128	0	141	77	64	0	141	39	102
Apple App Store Connect	0	48	18	30	0	48	6	42	2	46	0	48	34	14	0	48	12	36
Bitholic	0	6	1	5	1	5	0	6	0	6	0	6	3	3	0	6	2	4
BroadCom	0	40	8	32	4	36	12	28	2	38	1	39	37	3	0	40	9	31
GroupWise	0	135	33	102	2	133	19	116	0	135	0	135	105	30	0	135	18	117
Guardian News	0	5	0	5	1	4	0	5	4	1	0	5	0	5	0	5	5	0
SurveyJS	0	32	0	32	5	27	1	31	0	32	0	32	32	0	0	32	2	30
Percentage (in %)	8	92	18	82	3	97	10	90	5	95	0	100	71	29	0	100	21	79

analysis. The last row shows the percentage of end-points that are detected as antipatterns and patterns.

As shown in Table II for public APIs, almost all the analyzed end-points are tidy, i.e., 1,434 out of 1,443 end-points are well-designed in terms of syntactic structure. Similar observation can be made for contextualized resources (90%), verbless end-point (95%), descriptive end-point (94%), consistent documentation (93%), and standard end-point (92%). More specifically, end-points in public APIs generally (1) are designed with contextual resources names, (2) do not include CRUDy terms in their design, (3) are descriptive and easy to understand for human readers, (4) are not in contradiction with the HTTP method and the documentation, and (5) conform to standard character-set for end-point design.

Figures 2, 3, and 4 show the mosaic plots for the detection of nine linguistic patterns and antipatterns respectively in public, partner, and private APIs. In the figures, the white and black boxes represent the pattern and antipattern detection, respectively. The height of the boxes represents the number of end-points analyzed for an API. The width of the white and black boxes for each antipattern represents the ratio of end-points detected as patterns and antipatterns, respectively. As Figure 2 shows, all public APIs have less-cohesive documentation, i.e., ✖*Non-pertinent Documentation* is common among all public APIs. Some public APIs show a small proportion of contextless end-point design, e.g., Amazon AWS Core IoT

(9%), Facebook Graph API (11%), Samsung ARTIK Cloud (14%), Stack Overflow (12%). Moreover, all the public APIs except Samsung ARTIK Cloud have a standard end-point design. From the set of public APIs, Microsoft Azure IoT Hub and Dropbox, on average, have more antipatterns than others. As opposed, YouTube and Node-RED are among the APIs with higher design quality in terms of linguistic antipatterns.

For partner APIs in Table II, end-points are found to have contextless design, i.e., 50% of the analyzed end-points had ✖*Contextless Resource Names*. Also, a significant number of end-points (47%) were non-descriptive, and the majority of end-points (76%) have less cohesive documentation. Compared to the public APIs, the partner APIs have significantly less proportion of end-points having ✖*Inconsistent Documentation*, that is, only 1% of end-points in partner APIs have contradicting documentation whereas this was 7% for public APIs. Moreover, partner APIs do not use CRUDy terms in their end-points design, which is a good design practice. Overall, although seven out of nine linguistic antipatterns were found in nine partner APIs, in most cases, the partner APIs have a significantly higher proportion of patterns than the corresponding antipatterns. As Figure 3 shows, only Shopify has a higher proportion of linguistic antipatterns where it mostly lacks tidy, contextual, non-descriptive design for its end-points. Also, all partner APIs lack cohesive documentation, i.e., have ✖*Non-pertinent Documentation*. The partner APIs Shopify and QuickBooks Online are found, on average, to have more antipatterns than other partner APIs. In contrast, Oracle Cloud Marketplace Publisher and Uber are found to have the least antipatterns.

Finally, for private APIs, as shown in Table II, a fewer proportion of end-points have amorphous, CRUDy, non-standard design. More specifically, only 8% of end-points are detected as amorphous, 3% of the end-points have CRUDy terms, and all end-points were detected as standard design. Overall, eight out of nine linguistic antipatterns were detected in seven private APIs. It is notable that the private and partner APIs seem to have lower quality of documentation than the public APIs, i.e., 57% of the analyzed end-points in public APIs have ✖*Non-pertinent Documentation*, compared to 76% in partner APIs and 71% in private APIs. As shown in Figure 4, contextless end-point design and pluralized nodes are the most common design problems in private APIs.

Summary on RQ$_1$: Linguistic antipatterns are prevalent in public, partner, and private APIs. The two most common problems in public and private APIs are the contextless design of end-points and pluralized nodes. In addition, less-cohesive documentation is a common problem for all three categories of APIs.

B. API Category Prone to Poor Linguistic Design (RQ$_2$)

RQ$_2$ aims to find whether one category of API is more prone to poor linguistic quality. In order to answer RQ$_2$, we normalize the antipattern detection results for the public, partner, and private APIs and compare among the three groups of APIs. Since our set of analyzed APIs has variable-sized end-points, we normalize the antipatterns detection based on the total detected instances and the total number of end-points analyzed for each API. Table III presents the proportion of end-points that are detected as antipatterns for three categories of APIs. The last column shows the Normalized Antipatterns Rate (NAR). We rely on Welch's T-test where we observe the difference between the mean NAR values of the two groups with unequal sample size. The tests are performed at the significance level of 0.05. As Table IV shows, the difference between the mean NAR of the public and partner APIs is not large enough to be statistically significant. When comparing the public and private APIs, we found that the difference between the mean of the public and private APIs is large enough to be statistically significant. Finally, the difference between the mean of the partner and private APIs is not large enough to be statistically significant.

Summary on RQ$_2$: Private APIs lack linguistic design quality significantly higher than public APIs. The linguistic design quality of partner and public APIs on average differs, however, the difference is statistically insignificant.

C. Linguistic Patterns and Antipatterns Occur Most (RQ$_3$)

In RQ$_3$, we want to identify which linguistic patterns and antipatterns are more common in public, partner, and private APIs. As shown in Table III, for public APIs, two common end-points design issues are ✖*Non-pertinent Documentation* (57% of end-point) and ✖*Pluralized Nodes* (29%), and common good design practices include tidy and verbless end-point design. That is, only 0.7% of end-points are identified as ✖*Amorphous End-point* and less than 4% of end-points are detected as ✖*CRUDy End-point*.

For partner APIs, the two most common poor design practices are ✖*Non-pertinent Documentation* (76%) and ✖*Contextless Resources* (50%). In contrast, two notably common good design practices considered in partner APIs are verbless end-point (i.e., ✔*Verbless End-point*, 100%) and consistent documentation for end-points (i.e., ✔*Consistent Documentation*, 99%). In Table III, for private APIs, the two most common end-points design issues include ✖*Non-pertinent Documentation* (71%) and ✖*Pluralized Nodes* (21%) whereas common good end-point design practices for private APIs are ✔*Tidy End-point* (92%) and ✔*Verbless End-point* (97%).

Figure 5 shows the proportion of end-points that are detected as linguistic antipatterns in public, partner, and private APIs. Thus, regardless of APIs category, ✖*Non-pertinent Documentation*, ✖*Pluralized Nodes*, and ✖*Contextless Resources* seem to be three most common end-point design problems. As for good design practices, ✔*Verbless End-point*, ✔*Tidy End-point*, and ✔*Standard End-point* are common across the APIs categories. Note that end-points from three APIs categories are generally found to follow ✔*Hierarchical Nodes*.

Summary on RQ$_3$: APIs from the three categories appear to lack quality documentation, i.e., the documentation is not cohesive with the end-points. Consumers primarily rely on the

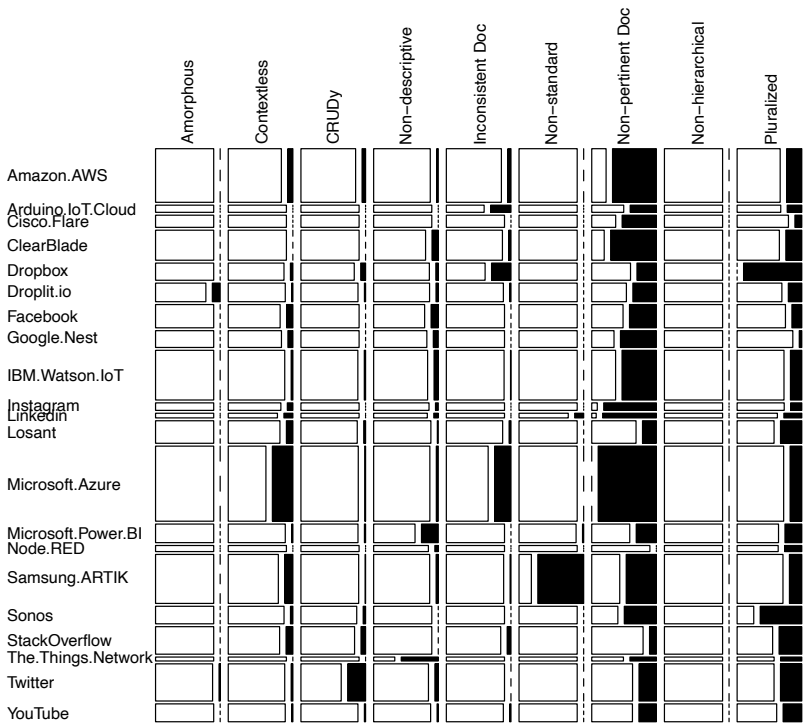

Fig. 2. Visualization of patterns/antipatterns in Public APIs. (Width of the black bars indicates the percentage of the endpoints suffering from that antipattern. The height of the API-box is the relative number of endpoints.)

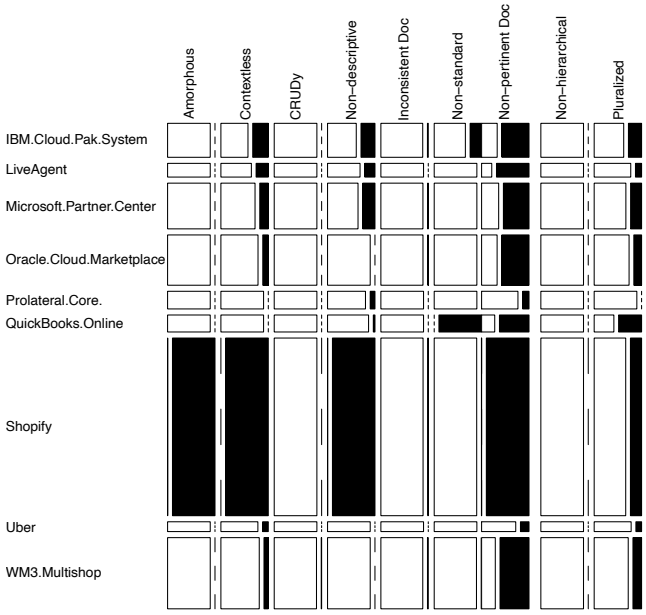

Fig. 3. Visualization of patterns/antipatterns in Partner APIs.

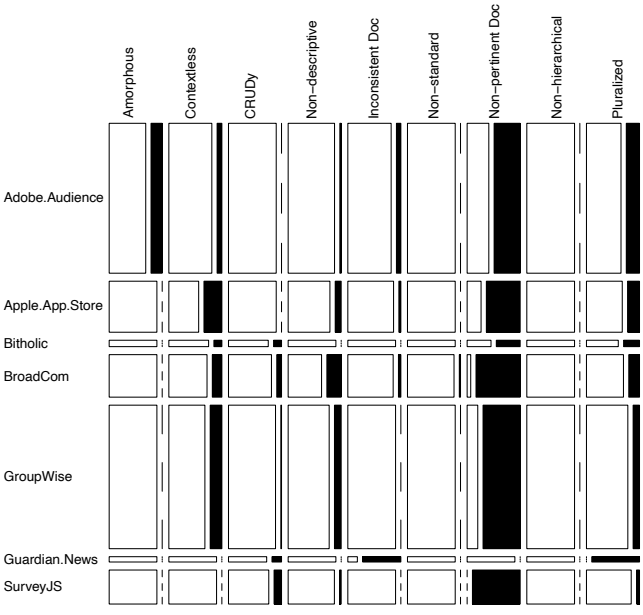

Fig. 4. Visualization of patterns/antipatterns in Private APIs.

API documentation to comprehend them, thus, API providers should consider quality API documentation essential.

V. Discussion

This section presents a general discussion on our observations from the APIs, the practical implications of our findings, and threats to validity of our results.

TABLE III
PERCENTAGE OF END-POINTS DETECTED AS ANTIPATTERNS IN 37 APIS.

APIs	✖Amorphous End-point	✖Contextless Resources	✖CRUDy End-point	✖Non-descriptive End-point	✖Inconsistent Doc.	✖Non-standard End-point	✖Non-pertinent Doc.	✖Non-hierarchical Nodes	✖Pluralized Nodes	**NAR**
Public APIs										
Amazon AWS Core IoT	0.0	8.7	6.0	2.7	5.3	0.0	75.3	0.0	26.0	13.8
Arduino IoT Cloud	0.0	0.0	0.0	0.0	35.0	0.0	45.0	0.0	25.0	11.7
Cisco Flare	0.0	0.0	0.0	0.0	0.0	0.0	58.8	0.0	11.8	7.8
ClearBlade	0.0	0.0	0.0	10.7	3.6	0.0	78.6	0.0	27.4	13.4
Dropbox	0.0	4.2	8.3	4.2	33.3	0.0	33.3	0.0	100.0	20.4
Droplit.io	13.5	1.9	1.9	3.8	1.9	0.0	40.4	0.0	23.1	9.6
Facebook	0.0	10.8	0.0	12.3	0.0	0.0	46.2	0.0	16.9	9.6
Google Nest	0.0	8.5	0.0	8.5	0.0	0.0	61.7	0.0	4.3	9.2
IBM Watson IoT	0.0	2.9	2.2	7.2	0.7	0.0	59.0	0.0	19.4	10.2
Instagram	0.0	9.5	0.0	4.8	0.0	0.0	90.5	0.0	19.0	13.8
Linkedin	0.0	15.4	0.0	7.7	0.0	15.4	92.3	0.0	30.8	17.9
Losant	0.0	11.1	3.2	1.6	3.2	0.0	23.8	0.0	36.5	8.8
Microsoft Azure IoT Hub	0.0	35.2	1.4	3.3	28.1	0.0	100.0	0.0	20.0	20.9
Microsoft Power BI	0.0	1.9	1.9	28.8	0.0	1.9	34.6	0.0	28.8	10.9
Node-RED	0.0	0.0	0.0	5.9	0.0	0.0	0.0	0.0	29.4	3.9
Samsung ARTIK Cloud	0.0	13.9	0.7	3.6	1.5	78.8	51.8	0.0	21.2	19.1
Sonos	0.0	4.1	4.1	0.0	0.0	0.0	55.1	0.0	71.4	15.0
Stack Overflow	0.0	11.7	7.8	0.0	6.5	0.0	11.7	0.0	39.0	8.5
The Things Network	0.0	0.0	0.0	63.6	0.0	0.0	45.5	0.0	36.4	16.2
Twitter	1.9	2.9	30.8	5.8	0.0	0.0	29.8	0.0	36.5	12.0
YouTube	0.0	2.0	2.0	2.0	2.0	0.0	30.0	0.0	32.0	7.8
Partner APIs										
IBM Cloud Pak System	0.0	36.5	0.0	32.7	1.9	26.9	63.5	0.0	30.8	21.4
LiveAgent	0.0	28.6	0.0	23.8	0.0	0.0	76.2	0.0	14.3	15.9
Microsoft Partner Center	0.0	20.0	0.0	28.6	1.4	0.0	60.0	0.0	25.7	15.1
Oracle Cloud Marketplace Publisher	0.0	13.0	0.0	0.0	1.3	0.0	63.6	0.0	18.2	10.7
Prolateral Core Infrastructure	0.0	0.0	0.0	11.1	0.0	0.0	14.8	0.0	0.0	2.9
QuickBooks Online	0.0	0.0	0.0	3.8	0.0	100.0	69.2	0.0	53.8	25.2
Shopify	99.6	100.0	0.0	99.6	1.1	0.4	100.0	0.0	26.1	47.4
Uber	0.0	13.3	0.0	0.0	0.0	0.0	20.0	0.0	13.3	5.2
WM3 Multishop	0.0	10.1	0.9	0.0	1.8	0.9	67.9	0.0	20.2	11.3
Private APIs										
Adobe Audience Manager	23.4	9.9	0.0	2.8	9.2	0.0	54.6	0.0	27.7	14.2
Apple App Store Connect	0.0	37.5	0.0	12.5	4.2	0.0	70.8	0.0	25.0	16.7
Bitholic	0.0	16.7	16.7	0.0	0.0	0.0	50.0	0.0	33.3	13.0
BroadCom	0.0	20.0	10.0	30.0	5.0	2.5	92.5	0.0	22.5	20.3
GroupWise	0.0	24.4	1.5	14.1	0.0	0.0	77.8	0.0	13.3	14.6
Guardian News	0.0	0.0	20.0	0.0	80.0	0.0	0.0	0.0	100.0	22.2
SurveyJS	0.0	0.0	15.6	3.1	0.0	0.0	100.0	0.0	6.3	13.9

TABLE IV
THE TWO-TAILED WELCH'S T-TEST BETWEEN THE GROUPS OF APIS.

Groups	*P-value*	Comments	Effect Size
Public APIs $\sim$ Private APIs	0.0286393	Significant at $p < 0.05$	large (0.93)
Public APIs $\sim$ Partner APIs	0.331575	Not significant at $p < 0.05$	medium (0.58)
Partner APIs $\sim$ Private APIs	0.884939	Not significant at $p < 0.05$	small (0.067)

A. API-specific Observations

Stack Overflow used HTTP POST methods for deleting REST resources using the URIs designed as /comments/{id} /delete or /questions/{id}/delete. In REST, DELETE method is recommended to delete resources, and the end-points should not be CRUDy. Terms related to create, read, update, and delete should not be used as part of the end-points. Instead, an appropriate HTTP method should be used, and the end-points should be designed accordingly. Moreover, Stack Overflow used only HTTP GET and POST methods for all sorts of actions on its resources, which is not a good REST practice and known as *Tunneling Everything through GET/POST* [17].

Shopify defined all its end-points using a .json extension in the form of https://partners.shopify.com/***/***.json, i.e., this signifies that the returned resources are by default represented in JSON. These are detected as ✖*Amorphous End-point*. In fact, there are more acceptable ways in REST to represent the type of resources as part of request and response

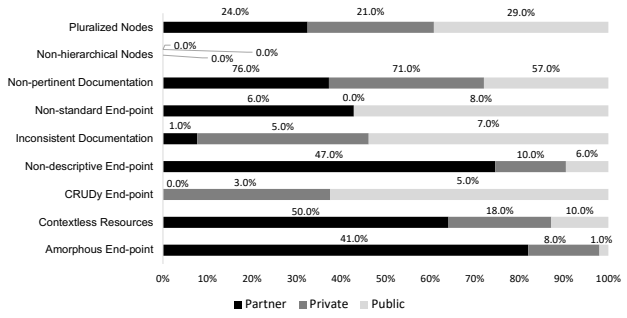

Fig. 5. Ratio of end-points detected as linguistic antipatterns in all APIs.

headers. Designing end-points with file extensions reduce their readability. Indeed, Shopify was identified as an outlier for some antipatterns detection. For example, all end-points from this API are detected as ✖*Non-Pertinent Documentation*. This is because the documentation, in most cases, are very brief, which may hinder comprehension of the purpose of end-points.

We found that only a handful of APIs leverage the set of available HTTP methods, e.g., Microsoft Partner Center uses the PATCH method on the end-point /customers/{customer_id}/DevicePolicyUpdates to update device policy. In addition, said API used the HEAD method to determine if a domain is available using the /domains/{domain} end-point. In REST, one uses the HEAD method to get response header meta-data instead of using GET method. There are also instances of misusing HTTP methods. For example, the Microsoft Partner Center used the POST method to retrieve inventory validation results for a country using the /extensions/product/checkinventory end-point. In REST, the POST method is not intended to retrieve resources. Seldom, APIs support dual methods. For example, IBM Watson provides end-point /deployment/resources/subnet/⟨subnetUUID⟩/addresses where consumers can use either POST or PUT method to create a range of addresses that are attached to a subnet.

B. Practical Implications

Publishing APIs as open and public brings more challenges for API providers. One of the main challenges is to ensure the best end-user experience when it comes to access information assets through APIs. Public APIs should be designed to be easily understandable and accessible to a wider group of Web and mobile developers. Thus, quality with simple design and documentation for APIs is essential. An API with quality design may be described from multiple perspectives. Among them linguistic design is key. Developers take a first glimpse at APIs and their documentation before they start using them. Therefore, a well-designed and documented API will facilitate the understandability and usability of the API. An API can be judged as well-designed when it has tidy end-points, has contextual design, and has cohesive and consistent documentation. This study aimed to observe whether three categories of APIs suffer from poor linguistic design. Indeed, all three categories of APIs lack quality design, measured through linguistic antipatterns. The significance of our findings lies in the benefits API providers will have by improving their APIs towards best practices (i.e., patterns) and avoiding poor practices (i.e., antipatterns). This, in turn, will facilitate APIs attracting more consumers.

In general, we observed that developers for public, partner, and private APIs do not always document their APIs well, i.e., the end-points and their documentation are not cohesive. Thus, a high portion of end-points are detected as ✖*Non-pertinent Documentation*. Developers should put more effort in documenting their APIs. Moreover, unlike in partner APIs, developers for public and private APIs tend to provide contradicting documentation, i.e., the underlying HTTP methods, the designed end-points, and their documentation are to some extent contradicting. This could be because the partner APIs are used by strategic business partners for which developers of partner APIs are strongly accounted for high-quality APIs.

Domain knowledge required by the API consumers and the *design context* of the API are the two most critical constraints that are distinct among the public, private, and partner APIs. For example, consumers for the private APIs have more domain knowledge, i.e., what their organization is doing, the data, and the services. Thus, in our study, oftentimes private APIs were found to have significantly higher occurrences of ✖*Non-pertinent Documentation* and ✖*Inconsistent Documentation*. Meaning that developers of private APIs tend to be less considerate in documenting their APIs. Moreover, concerning the design context, public APIs usually have no design context and are focused on one specific use case/application, and thus have fewer design constraints. Thus, developers of public APIs try to design *eye-catching* APIs with no or fewer constraints to attract as many clients. As a result, public APIs may often be of higher design quality. In contrast, private and partner APIs might require to match their design with existing APIs compromising the design quality as a whole. In our study, Shopify API was such an example, which was also an outlier.

C. Threats to Validity

Our findings may not be generalized to other APIs. To minimize threat to *external validity* of our results, we performed experiments on a set of more than 2,500 end-points from 37 APIs. However, further experiments are needed to confirm the findings. We relied on the SARA approach [6] and the SOFA framework [14] for detection of linguistic patterns and antipatterns. The approach and framework were effective in detecting linguistic antipatterns with an average precision and recall of more than 80%. The detection of linguistic antipatterns was performed on the APIs that had well-organized and self-contained documentation. Thus, detection on APIs with no or very minimal documentation could yield different results. However, we minimized threats to *construct validity* by selecting a well-documented set of APIs.

Moreover, we tried to minimize threat to construct validity by relying on the detection heuristics that are already shown to be effective in previous studies [6]. As for threats to *conclusion validity*, the false discovery rate adjusted p-value should be

0.017. Thus, our conclusions likely have type-1 error, i.e., we cannot be certain that there is a statistically significant difference between the groups of APIs. However, we tried to minimize this threat by using each dataset only two times in our tests. To minimize threats to *reliability* and *replicability validity*, we have put all APIs, their documentation, our detection results, and analysis results online[1].

VI. Related Work

Several studies analyzed REST APIs and detected patterns and antipatterns to assess their linguistic design. For example, studies in [6], [7], [8] assessed linguistic design of public APIs. These studies utilized syntactic and semantic analysis of end-points. To illustrate, the SARA approach performed the detection of linguistic patterns and antipatterns in public APIs. The authors in [6] studied twelve linguistic patterns and antipatterns and performed their detection on 18 public APIs, including LinkedIn, Facebook, Instagram, Twitter, and YouTube, with an average precision of over 80%.

In a study by Kotstein and Bodger [5], several design rules were rated in terms of both software quality and importance by industry experts from large IT enterprises. The main finding was that factors for quality were based on perceived usability, maintainability and compatibility. Aghajani et al. [7] did a large empirical study on the linguistic antipatterns in APIs on releases from a popular library, open-source Java projects and questions from Stack Overflow. The study demonstrated that only 2.6%-7% (all releases and latest releases) of API methods are used by web developers. An interesting finding is the correlation between documentation comments in public projects and the use of their methods, but without being able to define the causation clearly. They demonstrated that projects using APIs with linguistic antipatterns in the documentation had 29% higher probability for software bugs, which implies that linguistic antipatterns affect software quality.

Alshraideh and Katuk [8] proposed an algorithm that detected antipatterns in APIs. The results indicated that many of APIs were using an amorphous or unstructured end-points. The findings indicate the need to avoid amorphous end-points and ambiguous name patterns for developers to utilize and understand services. Panziera and Paoli [9] put forward a set of best practices for building self-descriptive REST services, which can be both human-readable and machine-processable, e.g., by using a common vocabulary for REST resources. They proposed a framework to collect information on documentation for generating descriptions of REST services, and evaluated the framework to identify resources correctly with precision and recall of 72% and 77%, respectively.

VII. Conclusion and Future Work

Application Programming Interfaces (APIs) facilitate communication among the Web services, middle-wares, frameworks, and libraries and with their clients. At present, REST is the industry-standard protocol for such interactions. Public, partner, and private are the three main categories of APIs [2]. REST APIs with good design quality facilitate their consumption. In the literature, *linguistic patterns* (i.e., best practices of REST API design) and *linguistic antipatterns* (i.e., poor REST API design practices to avoid) can be used to assess the design quality of the APIs [6], [7], [8]. Our study suggests API developers the areas that need more attention in order to improve their API design quality.

We assessed the linguistic design of 2,519 end-points in 37 different REST APIs looking for linguistic antipatterns. The APIs were divided into 21 public, 9 partner, and 7 private APIs with 1,433,669, and 407 end-points, respectively. Using this data, we answered the following research questions: $\mathbf{RQ}_1$ on the extent to which the public, partner, and private APIs suffer from poor linguistic design (linguistic antipatterns), $\mathbf{RQ}_2$ on finding the category of APIs more prone to poor linguistic design, and $\mathbf{RQ}_3$ on finding most occurring linguistic patterns and antipatterns in three categories of APIs.

In response to $\mathbf{RQ}_1$ and $\mathbf{RQ}_3$, we found mean rates of discovered antipatterns are 12.4% for public APIs, 17.2% for partner APIs, and 16.5% for private APIs. The most common antipattern detected is ✖*Non-Pertinent Documentation* where all three types of APIs showed over 50% of end-points are not well-documented (i.e., 57%, 76%, 71% for public, partner, and private APIs, respectively), c.f. Fig 5 and Table III. The second worst problem for public and private APIs was ✖*Pluralized Nodes* with 29% and 21%, respectively. For partner APIs, ✖*Contextless Resource Names* was the second most common problem with 50% of end-points affected. To answer $\mathbf{RQ}_2$, we conducted statistical tests to find whether there is a difference in the rate of linguistic antipatterns between APIs in the different groups. We found a small, significant difference between public and private APIs where public APIs have a slightly higher linguistic quality, as shown in Table IV.

From all data analyzed, we observed a slight trend towards lower quality in private APIs. This supports our claim that private APIs may have other design factors than public APIs and that other trade-offs are being made during development, e.g., development speed vs. linguistic design. However, it is unknown whether such trade-offs are a conscious decision or an effect of other factors such as fewer users, informal communication, or inexperienced developers.

Composite APIs combine two or more APIs to craft a sequence of interdependent operations. As part of our future plan, we also want to analyze and compare with another category of APIs, i.e., composite APIs. We can look at whether different categories of APIs from the same provider differ in their design. More investigation is required to find whether there are relations among the design issues, i.e., whether an introduction of one linguistic antipattern leads to introducing other related antipatterns. It would also be interesting to share our findings from different categories of APIs with the actual API developers and corroborate their design practices in place. We also plan to further generalize the findings from this research by analyzing more public, partner, and private APIs.

[1] https://doi.org/10.5281/zenodo.6603065

REFERENCES

[1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[2] D. Jacobson, G. Brail, and D. Woods, *APIs: A Strategy Guide*. O'Reilly Media, Inc., 2012.

[3] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A New Family of Software Anti-patterns: Linguistic Anti-patterns," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 187–196.

[4] M. Massé, "REST API Design Rulebook," in *Vasa*. O'Reilly, 2012, p. 114.

[5] S. Kotstein and J. Bogner, "Which restful api design rules are important and how do they improve software quality? a delphi study with industry experts," *arXiv preprint arXiv:2108.00033*, 2021.

[6] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns," *International Journal of Cooperative Information Systems*, vol. 26, no. 02, p. 1742001, 2017.

[7] E. Aghajani, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on linguistic antipatterns affecting apis," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 25–35.

[8] F. S. Alshraiedeh and N. Katuk, "A uri parsing technique and algorithm for anti-pattern detection in restful web services," *International Journal of Web Information Systems*, 2020.

[9] L. Panziera and F. D. Paoli, "A Framework for Self-descriptive RESTful Services," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2013, pp. 1407–1414.

[10] Microsoft MSDN, "Capitalization Styles," last visited: October 2021. [Online]. Available: https://msdn.microsoft.com/en-us/library/x2dbyw72(v=vs.71).aspx

[11] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, "Are restful apis well-designed? detection of their linguistic (anti) patterns," in *International Conference on Service-Oriented Computing*. Springer, 2015, pp. 171–187.

[12] T. Fredrich, "RESTful Service Best Practices: Recommendations for Creating Web Services," May 2012. [Online]. Available: http://www.restapitutorial.com/resources.html

[13] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic Antipatterns: What They Are And How Developers Perceive Them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, Feb 2016.

[14] F. Palma, N. Moha, and Y.-G. Guéhéneuc, "UniDoSA: The Unified Specification and Detection of Service Antipatterns," *IEEE Transactions on Software Engineering*, vol. 45, no. 10, pp. 1024–1053, 2018.

[15] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. Chapman and Hall/CRC, 2003.

[16] N. Cliff, "Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[17] C. Pautasso, "Some REST Design Patterns (and Anti-patterns)," 2009.