The Journal of Systems & Software 191 (2022) 111369

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Assessing the linguistic quality of REST APIs for IoT applications[☆]

Francis Palma[a,*], Tobias Olsson[a], Anna Wingkvist[a], Javier Gonzalez-Huerta[b]

[a] Department of Computer Science and Media Technology, Linnaeus University, Sweden
[b] Department of Software Engineering, Blekinge Institute of Technology, Sweden

ARTICLE INFO

Article history:
Received 9 January 2021
Received in revised form 30 March 2022
Accepted 12 May 2022
Available online 20 May 2022

Keywords:
REST APIs
IoT applications
Linguistic quality
Pattern
Antipattern
Detection

ABSTRACT

Internet of Things (IoT) is a growing technology that relies on connected 'things' that gather data from peer devices and send data to servers via APIs (Application Programming Interfaces). The design quality of those APIs has a direct impact on their understandability and reusability. This study focuses on the linguistic design quality of REST APIs for IoT applications and assesses their linguistic quality by performing the detection of linguistic patterns and antipatterns in REST APIs for IoT applications. *Linguistic antipatterns* are considered poor practices in the naming, documentation, and choice of identifiers. In contrast, *linguistic patterns* represent best practices to APIs design. The linguistic patterns and their corresponding antipatterns are hence contrasting pairs. We propose the SARAv2 (Semantic Analysis of REST APIs version two) approach to perform syntactic and semantic analyses of REST APIs for IoT applications. Based on the SARAv2 approach, we develop the REST-Ling tool and empirically validate the detection results of nine linguistic antipatterns. We analyse 19 REST APIs for IoT applications. Our detection results show that the linguistic antipatterns are prevalent and the REST-Ling tool can detect linguistic patterns and antipatterns in REST APIs for IoT applications with an average accuracy of over 80%. Moreover, the tool performs the detection of linguistic antipatterns on average in the order of seconds, *i.e.*, 8.396 s. We found that APIs generally follow good linguistic practices, although the prevalence of poor practices exists.

1. Introduction

The Internet of Things (IoT) is an emerging technology that relies on networks of 'things' – smart computing devices – communicating with each other over the Internet. These 'connected things' (*i.e.*, IoT devices) frequently gather data and send them to servers or receive data from peer devices and act on them (Madakam et al., 2015).

REpresentational State Transfer (REST) (Fielding, 2000) is the *de facto* standard to design, develop, and deploy IoT-based applications in cloud environments. The application programming interfaces (APIs) for IoT applications are designed following the REST principles by the IoT vendors. The Message Queuing Telemetry Transport (MQTT) protocol is also supported by some APIs and the topic structure of MQTT could be seen as an alternative to the API design in REST. However, the offered functionality is often more limited in its scope, and it is often only recommended to use MQTT for certain applications where the lightweight nature of the protocol is necessary. In this article, for the sake of simplicity, from this point on, we refer to the 'REST APIs for IoT applications' as 'IoT APIs'. Client developers design and develop applications for IoT devices using vendor-provided IoT APIs that interact/communicate with peers and gateway servers. The design quality of IoT APIs has a direct impact on their understandability and reusability. Well-designed and named APIs may attract client developers more than poorly designed or named APIs (Massé, 2012) because they must understand the providers' APIs while integrating their services.

In previous works, we have performed analyses on the APIs for Web applications (*e.g.*, Facebook, YouTube, Dropbox, etc.) and Cloud services (*e.g.*, Open Stack) (Palma et al., 2014, 2015; Petrillo et al., 2016). Yet, no such study has been performed to investigate how well the APIs dedicated to the IoT applications are designed in terms of linguistic quality. To measure the linguistic quality, we perform syntactic and semantic analysis of the URIs and their documentation. According to Wilhelm et al. (2013), in the context of computer programs, syntactic analysis recognises the syntactic structure of the programs. In contrast, semantic analysis helps determine properties and check conditions relevant to the programs' well-formedness according to the programming language rules. Thus, in our study, the syntactic analysis concerns the syntactic structure of the resource URIs and the semantic analysis

☆ Editor: Shane McIntosh.

∗ Corresponding author.
E-mail addresses: francis.palma@lnu.se (F. Palma), tobias.olsson@lnu.se (T. Olsson), anna.wingkvist@lnu.se (A. Wingkvist), javier.gonzalez.huerta@bth.se (J. Gonzalez-Huerta).

https://doi.org/10.1016/j.jss.2022.111369
0164-1212/

checks for the well-formedness of the resource URIs according to the good API design practices defined in the literature (Massé, 2012; Petrillo et al., 2016, 2018a; OMA, 2012; Petrillo et al., 2018b; Haupt et al., 2017; Hausenblas, 2011; Brabra et al., 2016). We perform a study on APIs solely designed for IoT devices and applications.

The linguistic and semantic relations among the 'things', services, and parameters are as crucial in IoT APIs as in APIs for Web applications (Fredrich, 2012). The lack of such relations and/or poor naming may degrade the overall design of IoT APIs and translate into *linguistic antipatterns*. In the context of IoT APIs, linguistic antipatterns are poor solutions to common URI (Uniform Resource Identifier) design problems, which may hinder the consumption and reuse of IoT APIs by client developers; and the maintenance and evolution of IoT APIs by API vendors. Conversely, linguistic patterns represent good solutions to common URI design problems and facilitate the consumption and maintenance of IoT APIs. Thus, the linguistic patterns and their corresponding antipatterns are contrasting pairs.

An example of a poor practice is ✖*Inconsistent Documentation*[1] where a resource URI (together with the HTTP method) is in contradiction with its documentation. In the IBM Watson IoT, the POST method with `/bulk/devices/remove` URI is in contradiction with its documentation.[2] In REST, the POST method should be used to create something. The presence of *Inconsistent Documentation* may confuse IoT client developers who require clear and uniform resource specifications. The understandability and usability of the API might be hindered if this linguistic antipattern exists. In contrast, ✔*Consistent Documentation*[3] is a linguistic pattern where a URI is in line with its documentation. The URI `/draft/physicalinterfaces/{physicalInterfaceId}` with the HTTP DELETE method from the IBM Watson IoT API is an example of this pattern with its documentation.[4] In this research, we propose the SARAv2 approach (Semantic Analysis of REST APIs version two) as an extension to our previous approach, SARA (Palma et al., 2017). SARAv2 is can perform semantic analysis of REST APIs in general, and therefore also IoT REST APIs, aiming to assess their linguistic quality, by detecting linguistic patterns and antipatterns. Being inspired from the object-oriented domain (Arnaoudova et al., 2013, 2016), we define three new linguistic patterns and their corresponding linguistic antipatterns, namely *Consistent vs. Inconsistent Documentation*, *Versioned vs. Unversioned URIs*, and *Standard vs. Non-standard URI*.

We develop the REST-Ling tool as the implementation of the SARAv2 approach. REST-Ling is a web application that automates the detection of linguistic patterns and antipatterns. Applying the REST-Ling tool, we perform the detection of nine linguistic patterns and their corresponding antipatterns in 1102 URIs from 19 IoT APIs, *e.g.*, Amazon, Cisco, Google, IBM, Microsoft, Samsung. REST-Ling utilises various NLP techniques including the traditional WordNet (Fellbaum, 1998) and Stanford's CoreNLP (Manning et al., 2014) general-purpose English dictionaries analysed with Latent Dirichlet Allocation (LDA) (Blei et al., 2003) topic modelling technique and benefit from the second-order semantic similarity metrics (Kolb, 2008, 2009).

In summary, our five key contributions are:

1. the SARAv2 approach – an extension of SARA (Palma et al., 2017) – for the syntactic and semantic analysis of REST APIs for IoT applications;
2. the definitions of three new linguistic patterns and antipatterns and their detection algorithms;
3. an empirical assessment of the linguistic quality of a set of 19 IoT APIs from 18 different IoT providers;
4. a web-based tool, REST-Ling available on https://rest-ling.com for the detection of linguistic and structural antipatterns and patterns;
5. the empirical validation of the REST-Ling tool focusing on its accuracy and efficiency;
6. a comparison with relevant studies on the detection of linguistic patterns and antipatterns from other domains (*i.e.*, APIs for Cloud services and Web applications).

We also perform a comparison with relevant studies on the detection of linguistic patterns and antipatterns from other domains (*i.e.*, APIs for Cloud services and Web applications) from the perspectives of antipatterns and various accuracy measures. To assess the linguistic quality of IoT APIs and validate the SARAv2 approach, we define and answer the following four research questions:

- RQ_1 Prevalence: *To what extent IoT APIs suffer from poor linguistic design quality, i.e., linguistic antipatterns?*
- RQ_2 Comparison: *To what extent APIs across domains suffer from poor linguistic design quality, i.e., linguistic antipatterns?*
- RQ_3 Accuracy: *What is the accuracy of REST-Ling on the detection of linguistic antipatterns?*
- RQ_4 Efficiency: *How does the REST-Ling perform in terms of average detection time for linguistic antipatterns?*

Our empirical results show that (1) out of the 19 analysed IoT APIs, only a few of them have syntactic design problems and most of the analysed URIs follow good linguistic practices, although there also exist certain poor practices in some specific APIs. Examples include: ✖*Non-pertinent Documentation* was common in all IoT APIs and majority of the APIs had ✖*Unversioned URI* antipattern. In contrast, almost all of the APIs followed ✔*Tidy URI* and ✔*Consistent Documentation* patterns; and (2) the REST-Ling tool has an average accuracy over 80% when analysing IoT APIs.

The remaining article is structured as follows: Section 2 describes the linguistic patterns and antipatterns studied. Section 3 presents the SARAv2 approach we apply for the detection of linguistic patterns and antipatterns. Section 4 shows experimental details and discusses the obtained detection results. Section 5 discusses related works and makes a comparison with other state-of-the-art studies. Finally, in Section 6 we conclude the research and present future work.

2. Linguistic patterns and antipatterns

In total, we gathered nine linguistic patterns and antipatterns. The first six patterns and antipatterns are from the literature on REST APIs (Massé, 2012; Palma et al., 2015; Fredrich, 2012; Palma et al., 2017; Berners-Lee et al., 2005; Tilkov, 2008) and the final three antipatterns are newly defined in this study.

To define new linguistic antipatterns, we studied similar linguistic antipatterns that exist in the object-oriented literature (e.g., related to class or method signature and source code comments), and performed a data analysis by also looking at the URIs and API documentation in our API dataset, to see the applicability of those linguistic antipatterns, and created "themes" of patterns and antipatterns that are applicable to REST URIs and documentation, by using thematic analysis (Braun and Clarke, 2006). We adapted the detection heuristics from the object-oriented domain to the context of APIs that have resource identifiers (i.e., URIs) and their documentation. We defined ✖*Inconsistent Documentation* linguistic antipattern being inspired from Arnaoudova et al.

[1] We use the ✖ symbol to refer an *antipattern*.

[2] `Delete multiple devices. Delete multiple devices, each request can contain a maximum of 512 kB.`

[3] We use the ✔ symbol to refer a *pattern*.

[4] `Delete a draft physical interface. Deletes the draft physical interface with the specified id from the draft physical interface.`

(2016). We also studied the gray literature to discover concerns from the practitioners and formalise those observations in the form of linguistic antipatterns and their corresponding patterns. For example, the concept of ✖*Unversioned URI* antipattern was discussed in Levin (2017). Another newly defined antipattern ✖*Non-standard URI Design* is defined based on the notion similar to ✖*Amorphous URI* antipattern (which affects the readability of the URIs) that non-standard characters should not be used in the URI design.

We formulated the detection heuristics of new linguistic antipatterns and patterns after a thorough discussion with the team consisting of two authors (who are not part of the manual validation). In the case of disagreement between the authors, a third opinion was sought from a researcher who also is not part of the experiment and validation. This enabled us to resolve the conflicts and avoid the bias by a specific author in defining new linguistic antipatterns and their detection heuristics. These new patterns and antipatterns are also applicable to APIs for Web applications or cloud services. The following subsections summarise the linguistic patterns and antipatterns SARAv2 can detect in REST APIs.

2.1. Tidy vs. Amorphous URIs

The URIs in REST should be tidy and easy to read. A *Tidy URI* has an appropriate lower-case resource naming, no extensions, underscores, or trailing slashes. *Amorphous URI* occurs when URIs contain symbols or capital letters that make them difficult to read and use. A URI is amorphous if it contains: (1) upper-case letter (except for Camel Cases Microsoft MSDN (2006)), (2) file extensions, (3) underscores, and, (4) a final trailing-slash (Massé, 2012; Palma et al., 2015). The URI www.exampleAlbum.com/NEW_Customer/image01.tiff/ is a ✖*Amorphous URI* since it includes a file extension, upper-case resource names, underscores, and a trailing slash. In contrast, the URI www.example.com/customers/1234 is a ✔*Tidy URI* since it only contains lower-case resource naming, without extensions, underscores, or trailing slashes. The detection of this design practice requires syntactic analysis of the URIs.

2.2. Contextualised vs. Contextless Resource Names

URIs should be *contextual*, *i.e.*, nodes in URIs should belong to semantically-related context. Thus, the *Contextless Resource Names* appears when URIs are composed of nodes that do not belong to the same semantic context (Fredrich, 2012). The URI www.example.com/newspapers/planet/players?id=123 is a ✖*Contextless Resource Names* because 'newspapers', 'planet', and 'players' do not belong to same semantic context. In contrast, the URI www.example.com/soccer/team/players?id=123 is a ✔*Contextual Resource Names* because 'soccer', 'team', and 'players' belong to same semantic context. The detection of *Contextualised vs. Contextless Resource Names* requires semantic analysis of the URIs.

2.3. Verbless vs. CRUDy URIs

Appropriate HTTP methods, *e.g.*, GET, POST, PUT, or DELETE, should be used in *Verbless URIs* instead of using CRUDy terms (*e.g.*, create, read, update, delete, or their synonyms) (Fredrich, 2012). The use of such terms as resource names or requested actions is highly discouraged (Massé, 2012; Fredrich, 2012). This URI with the HTTP POST www.example.com/update/players/age?id=123 is a ✖*CRUDy URIs* since it contains a CRUDy term 'update' while updating the user's profile colour relying on an HTTP POST method. In contrast, this URI with the HTTP method POST www.example.com/players/age?id=123 is a ✔*Verbless URIs* making an HTTP POST request without any verb. The detection of this design practice requires semantic analysis of the URIs.

2.4. Hierarchical vs. Non-hierarchical Nodes

Nodes in a URI should be hierarchically related to its neighbour nodes. In contrast, *Non-hierarchical Nodes* is an antipattern that appears when at least one node in a URI is not hierarchically related to its neighbour nodes (Fredrich, 2012). The URI www.examples1.com/professors/faculty/university is a ✖*Non-hierarchical Nodes* since 'professors', 'faculty', and 'university' are not in a hierarchical relationship. In contrast, the URI www.examples2.com/university/faculty/professors is a ✔*Hierarchical Nodes* since 'university', 'faculty', and 'professors' are in a hierarchical relationship. The detection of *Hierarchical vs. Non-hierarchical Nodes* requires semantic analysis of the URIs.

2.5. Singularised vs. Pluralised nodes

URIs 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 URI should be singular. In contrast, for POST requests, the last node should be plural. Therefore, the *Pluralised Nodes* antipattern appears when plural names are used for PUT/DELETE requests or singular names are used for POST requests. However, GET requests are not affected by this antipattern (Fredrich, 2012; Palma et al., 2015). The first example URI is a POST method that does not use a pluralised resource, thus leading to ✖*Pluralised Nodes*. In contrast, in the second example as shown below, for the ✔*Singularised Nodes*, the DELETE request acts on a single resource for deleting it. An example of ✖*Pluralised Nodes* is DELETE www.example.com/team/players or POST www.example.com/team/player. The ✔*Singularised Nodes* can be exemplified as DELETE www.example.com/team/player or POST www.example.com/team/players. The detection of this design practice requires semantic analysis of the URIs.

2.6. Pertinent vs. Non-pertinent documentation

The ✖*Non-pertinent Documentation* occurs when the documentation of a REST resource URI is in contradiction with its structure (*e.g.*, nodes separated by slashes in URIs), inspired from a similar antipattern from the OO domain (Arnaoudova et al., 2016). This antipattern applies to both a resource URI and its corresponding documentation. In contrast, a well-documented URI should properly and clearly describe its purpose using semantically related terms (Arnaoudova et al., 2016; Petrillo et al., 2018a). The URI-documentation pair from Twitter: `api.twitter.com/1.1/favourites/list` – '*Returns the 20 most recent Tweets liked by the authenticating or specified user*' shows no semantic similarity between them and, thus, considered as a ✖*Non-pertinent Documentation*. In contrast, this URI-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.*' shows a high relatedness and considered as a ✔*Pertinent Documentation*. The detection of this design practice requires semantic analysis of the URIs and their documentations.

2.7. Consistent vs. Inconsistent documentation

The ✖*Inconsistent Documentation* found in REST API documentation is defined based on another antipattern *Method Signature and Comment are Opposite* (Arnaoudova et al., 2016) common in object-oriented systems. It occurs if the documentation of a method is in contradiction with its declaration. REST API documentations may also manifest similar practice where a resource URI (together with the HTTP method) is in contradiction with

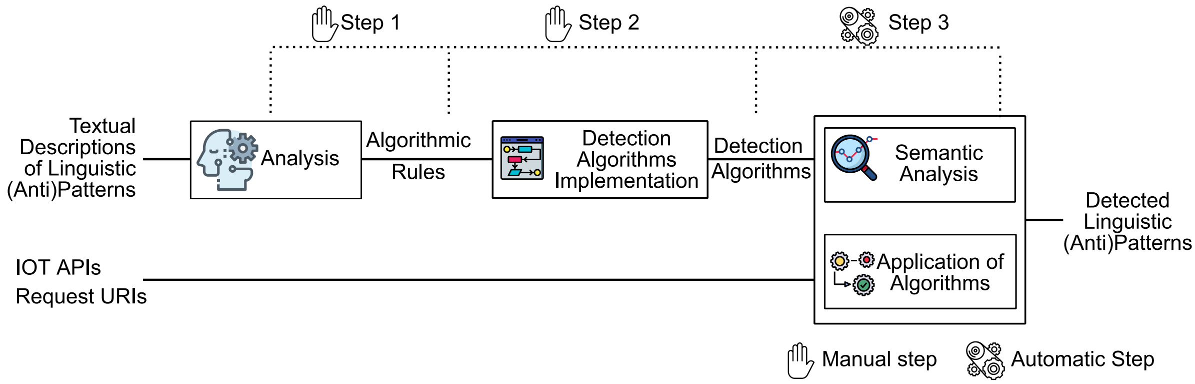

Fig. 1. Overview of the SARAv2 approach.

its documentation. For example, in the IBM Watson IoT, the POST method with `/bulk/devices/remove` URI is in contradiction with its documentation '*Delete multiple devices. Delete multiple devices, each request can contain a maximum of 512 kB*', thus, is an ✖*Inconsistent Documentation*. By REST design principles, the POST method should be used to create something. When a resource URI (together with the HTTP method) is in contradiction with its documentation. For the same example URI, `/bulk/devices/remove`, if the documentation were stated as '*Remove multiple devices. Remove multiple devices, each request can contain a maximum of 512 kB*', this could be identified as ✔*Consistent Documentation*. The detection of *Consistent vs. Inconsistent Documentation* requires semantic analysis of the URIs and their documentations.

2.8. Versioned vs. Unversioned URIs

APIs evolve continuously and if not properly versioned, might cause clients to break. Versioned APIs facilitates easy maintenance both for API providers and client developers. Changes in the APIs may include a change in the response data format or type, removing a resource, adding a new end-point, response parameters, which require to track major or minor versions for APIs. If an API is not versioned at all, the *Unversioned URI* linguistic antipattern occurs (Levin, 2017). For example, Losant API does not have any version information in its URI, whereas IBM Watson IoT always use versioned URI, thus follow the *Versioned URI* pattern (Levin, 2017). The URI `api.example.com/1.1/resourceid/view` is an example of ✔*Versioned URI* with its API version embedded in the URI. In contrast, the URI `api.example.com/resourceid/view` is an example of ✖*Unversioned URI*. The detection of this design practice requires syntactic analysis of the URIs.

2.9. Standard vs. Non-standard URI

The URI design should not include nodes or resources with non-standard identification, which hinders the reusability and understandability of the APIs. The ✖*Non-standard URI Design* occurs when (1) characters like é, å, ö, etc. are present in URIs, (2) blank spaces are found in URIs, (3) double hyphens are used in URIs, and (4) unknown characters (*e.g.*, !, @, #, $, %, ^, &, *, etc.) are present in URIs. Instead, a URI following ✔*Standard URI 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. The URI `api.example.com/museum/louvre/réception/` is an example of ✖*Non-standard URI Design*. While, the URI `api.example.com/museum/louvre/reception/` represents ✔*Standard URI Design*. The first example format hinders the usability and understandability as compared to the latter URI. The detection of *Standard vs. Non-standard URI* requires syntactic analysis of the URIs.

3. The SARAv2 approach

SARAv2 (Semantic Analysis of REST APIs version two) enables the automatic detection of nine linguistic antipatterns and their corresponding patterns in REST APIs and, therefore, in REST APIs for IoT. To analyse the REST APIs and their documentation, we manually collect a subset of URIs and their documentation provided by each REST API provider (*i.e.*, in this paper IoT providers). These collected URIs and their documentation are used later in the detection phase. As shown in Fig. 1, the SARAv2 approach consists of three steps:

Step 1. Analysis of Linguistic Patterns and Antipatterns: A manual step that consists of analysing the description of linguistic patterns and antipatterns from the literature to identify the properties relevant to their detection. We use these relevant properties to define *detection heuristics* for patterns and antipatterns.

Step 2. Implementation of Detection Algorithms: A second manual step that involves the implementation of concrete detection algorithms for patterns and antipatterns based on the detection heuristics defined in Step 1.

Step 3. Detection of Linguistic Patterns and Antipatterns: An automatic step that executes the semantic analysis of resource URIs and API documentation by automatically applying the detection algorithms (implemented in Step 2) on URIs and APIs documentation for the detection of linguistic patterns and antipatterns.

In the following sections, we discuss each step in SARAv2 in detail.

3.1. Analysis of linguistic patterns and antipatterns

We analyse the definitions of antipatterns and patterns defined in Section 2 to identify their various linguistic aspects. For example, a linguistic aspect for the detection of the ✖*Contextless Resource Names* is to assess whether a pair of URI nodes are semantically related, *i.e.*, belong to the same semantic context. Fig. 2 shows the detection heuristic for the ✖*Contextless Resource Names*. We extract the domain knowledge from the URI documentation and the request URI (lines 2–3), to build a topic model. We then calculate and check the similarity among the nodes (line 4). We check this similarity using the topic model we generate by applying natural language processing techniques. We calculate the average similarity value for all the nodes in a URI against each topic from our topic model. And, we report a URI has the ✖*Contextless Resource Names* if the average similarity value is less than the threshold (line 5). On the contrary, an occurrence of ✔*Contextual Resource Names* will be reported if the similarity value is equal to or higher than the threshold. Based on our previous studies (Palma et al., 2017) and findings by Kolb

```
Contextless-Resource(Request-URI, API-Documentation)
  TopicsModel ← Extract-Topics(API-Documentation)
  URINodes ← Extract-URI-Nodes(Request-URI)
  Similarity-Value ←Calculate-Second-Order-Similarity(URINodes, TopicsModel)
  if Similarity-Value < threshold
    return true ‘‘Contextless Resource Names" antipattern
  end if
  return false ‘‘Contextual Resource Names" pattern
```

Fig. 2. Detection heuristic for *Contextless Resource Names* antipattern.

```
1:  Inconsistent-Documentation(HTTP-Method, Request-URI, Documentation)
2:    Documentation ← Remove-Stop-Words(Documentation)
3:    Tokens ← Lemmatise-Tokenise(Documentation)
4:    if HTTP-Method = ‘POST’ && Synonyms(Delete or Update or Get) ∈ Tokens
5:      return true ‘‘Inconsistent Documentation" antipattern
6:    else if HTTP-Method = ‘DELETE’ && Synonyms(Create or Update or Get) ∈ Tokens
7:      return true ‘‘Inconsistent Documentation" antipattern
8:    else if HTTP-Method = ‘PUT’ && Synonyms(Create or Delete or Get) ∈ Tokens
9:      return true ‘‘Inconsistent Documentation" antipattern
10:   else if HTTP-Method = ‘GET’ && Synonyms(Delete or Update or Create) ∈ Tokens
11:     return true ‘‘Inconsistent Documentation" antipattern
12:   end if
13:   return false ‘‘Consistent Documentation" pattern
```

Fig. 3. Detection heuristic for *Inconsistent Documentation* antipattern.

(2009), we used 0.3 as the threshold to determine semantic relatedness between words. We empirically determined the threshold value, *i.e.*, we started from 0.1 and increased 0.05 each time the semantic relatedness for a set of pair of nodes is not reasonable. Moreover, on using DISCO, Kolb (2008, 2009) determined the threshold value of 0.3 can be utilised as the *gold standard* with good accuracy with regard to semantic relatedness.

Similarly, a linguistic aspect for the detection of the ✖*Inconsistent Documentation* is to assess whether the HTTP method used with a resource URI is described with a conflicting documentation. Fig. 3 presents the detection heuristic for the ✖*Inconsistent Documentation*. We begin with preprocessing the documentation by removing the stop words (line 2) and tokenise the documentation, i.e., obtain the set of words in the documentation, and lemmatise them, *i.e.*, we extract the base form of each word in the documentation (line 3). Then, we match with the HTTP method to check if the URI and its documentation are related and consistent (lines 4 to 13). To measure relatedness between the HTTP method and the documentation, we check whether the synonyms of various actions or verbs are misplaced within the documentation. For example, the HTTP POST method is often used to create a new resource if the resource does not exist already. Thus, the documentation related to this action or the resource on which the POST action is taken, must not have any indication of resource deletion, retrieval, or update. If such contradiction is found in the documentation then SARAv2 will report a ✖*Inconsistent Documentation* (lines 4–5). In contrast, an occurrence of ✔*Consistent Documentation* will be reported if no contradiction is discovered between the HTTP method/action and the documentation. The detection heuristics of other linguistic patterns and antipatterns are available online.[5]

3.2. Implementation of detection algorithms

To detect the linguistic pattern and antipattern, we implemented the detection algorithms using Java since our detection framework, SOFA (Service Oriented Framework for Antipatterns) (Palma et al., 2017), is Java-based. The SARAv2 approach does not require the parameterised URIs to perform the analysis, *i.e.*, performs the analysis on the URIs from the IoT APIs documentation. We manually convert (write the Java code) the detection heuristics defined in Section 3.1 into executable Java programs. Listing 1 shows an example of a code snippet in the form of pseudocode that we apply for the detection of ✖*Contextless Resource Names*.

As shown in Listing 1, once the `detectContextlessResource()` method is invoked (line 4) the `URIContextualAnalysis()` procedure is initiated (line 9), which is the implementation of the heuristics for *Contextless Resource Names* antipattern in Fig. 2. Inside the `URIContextualAnalysis()` procedure, first, the topic model is built (line 18), followed by the extraction of the nodes in the URI (line 20). We use a matrix for storing the similarity values between each node and the members in the topic model (line 22). Finally, the calculation of second-order similarity values takes place (lines 24 and 25), and the detection of either an antipattern or a pattern is decided based on the average similarity value for each node. If the average similarity value for all the nodes in a URI is below a predefined threshold (lines 27 to 30), we consider it as a contextless URI design, thus, an antipattern, and if above, it is considered as a contextual URI design, *i.e.*, a pattern.

3.3. Detection of linguistic patterns and antipatterns

In SARAv2, the detection of linguistic patterns and antipatterns utilises two essential elements: the Second Order Semantic Similarity metric and Latent Dirichlet Allocation (LDA).

The Second Order Semantic Similarity metric (Kolb, 2008, 2009) allows obtaining the distributionally most similar words for a given word, and computes similarity scores among them based on second-order word vectors. Two words are considered distributionally similar if they have multiple co-occurring words in the same syntactic relations (Budanitsky and Hirst, 2006). The distributional semantic similarity goes beyond *is-a* relationships

5 http://sofa.uqam.ca/santa/.

Listing 1: Code snippet for the detection *Contextless Resource Names* linguistic antipattern.

```
public static void main(String[] args) {
    ...
    /* Detection of Contextless Resource Names Antipattern */
    detectContextlessResourceNames();
    writeOutput();
}
private static void detectContextlessResource() {
    result = restAnalyser.URIContextualAnalysis(URI);
    if(!result)
        contextlessResultAP.add(URI);
    else
        contextlessResultP.add(URI);
}
public boolean URIContextualAnalysis(String Uri) {
    ...
    Topics = this.ldaProcessor.getTopicList();
    ...
    ArrayList<String> UriNodes = getUriNodes(Uri);
    ...
    for each Node in UriNodes and for each topic in Topics;
        Calc_Second_Order_Sim_Values();
    Calc_Avg_Sim_of_Nodes_in_URI();
    if average_similarity < threshold
        return true;
    else
        return false;
}
```

between nouns and verbs as allowed by approaches (Budanitsky and Hirst, 2006; Kolb, 2009) based on WordNet (Fellbaum, 1998) that only benefits from the synonym (warm-hot), meronym (car-wheel), and antonym (hot-cold) relations. Distributional semantic similarity captures the multiple senses of a given word and allows mixing all the distributionally similar semantic words for all these senses.

LDA is a generative probabilistic model of a corpus based on topic models. It relies on the idea that a document is a mixture of latent topics, and each topic is a probabilistic distribution over words (Blei et al., 2003) and supports the extraction of topic models from a corpus. The topic model is a low-dimensional representation of the content of the documents in the corpus. LDA allows a document to pertain to many different topics by associating the probability of the document belonging to each topic, overcoming one of the main problems of many other clustering models that restrict documents to be associated with just one topic. LDA is also affected by the *bag-of-words* assumptions, meaning that words that appear or should be generated by a topic might also be allocated in other topics (Blei et al., 2003). To tackle these problems, we defined a hybrid approach for SARA (Palma et al., 2017), combining LDA topic modelling to obtain the low-dimensional representation of the corpus and the distributional semantic similarity to measure the semantic similarity between the words.

3.3.1. Semantic analysis of IoT APIs

SARAv2 strategy for performing the semantic analysis of IoT APIs involves four automatic steps, as illustrated in Fig. 4: (1) collecting IoT APIs documentation and their preprocessing (*i.e.*, exclusion of stop words); (2) truncating example URI nodes to their base form (*i.e.*, lemmatisation) using Stanford's CoreNLP (Manning et al., 2014); (3) extracting the LDA topic model using the collected corpora; and (4) measuring the second-order similarity between the extracted LDA topic model and the URI nodes. The LDA topic model is created by using the Mallet LDA topic modelling tool-set[6] using the IoT APIs documentation, excluding lists

[6] http://mallet.cs.umass.edu.

Table 1
Top 15 words for Google Nest topic model with k = 3.

Topic 1	Topic 2	Topic 3
eco	smoke	device
record	sound	structure
estimate	snapshot	thermostat
lock	status	event
adjust	change	nest
format	list	camera
related	display	url
json	nest	display
call	expire	temperature
sign	home	alarm
home	detect	require
sound	subscription	hvac
bandwidth	field	aware
low	live	zone
image	motion	activity

of parameters, and response formats. This LDA model represents a minimal representation of the members of the corpus, preserving the essential semantic relationships needed for classification (Blei et al., 2003).

In the following, we briefly describe how we determine the detection of a linguistic antipattern *Contextless Resource Names* (and the corresponding *Contextual Resource Names* pattern) using the LDA topic modelling (Blei et al., 2003) and second-order semantic similarity (Kolb, 2008, 2009).

3.3.2. Determining patterns and antipatterns

To discover the relationships (*e.g.*, contextual) between the pair of nodes (*i.e.*, resource identifiers) in the URIs, as mentioned above, we use the tool-set based on Mallet LDA topic modelling. Given a collection of text (or documents), LDA generates a topic model that specifies the important relationships crucial for classification or summarisation tasks (Blei et al., 2003). In other words, the generated topic model represents the collection of documents in some low-dimensional word vectors. The topic model for each IoT API was built after gathering the descriptions of the resource URIs as input by excluding the list of parameters, request or response formats, and example code.

We start processing the collection of text by removing the stop words and expanding the acronyms to get the full form. For this, we collect a list of API-specific acronyms. The collection of acronyms is performed after we gather 1102 URIs and their documentation for 19 IoT APIs. We go through each URI and its documentation, look for acronyms and create an API-specific dictionary that includes the acronyms and their full forms. Later, during the processing of the URIs and their documentation, we replace the acronyms with their full forms to build a more accurate topic model. The lemmatisation process is also applied to set the words to their base form, for which we rely on Stanford CoreNLP (Manning et al., 2014). Then, we obtain the topic model with k topics using the Mallet tool-set for which the set of unique end-points for an IoT API is considered as topics. This is done because end-points are the key concepts for an API as they appear first in the URI design hierarchy (Fredrich, 2012).

Table 1 shows the topic model obtained using Mallet from the documentation corpus of Google Nest IoT API. As shown, the topic model has three topics, and for each topic, the 15 most relevant words are listed. Later, we use this topic model to quantify the similarity between a pair of nodes in a URI. For example, two nodes (or resource identifiers) are related or similar if they belong to the same topic following the method proposed by Griffith and Steyvers (2006).

After building the LDA topic model, we rely on the second-order semantic similarity metric to compute the (semantic or

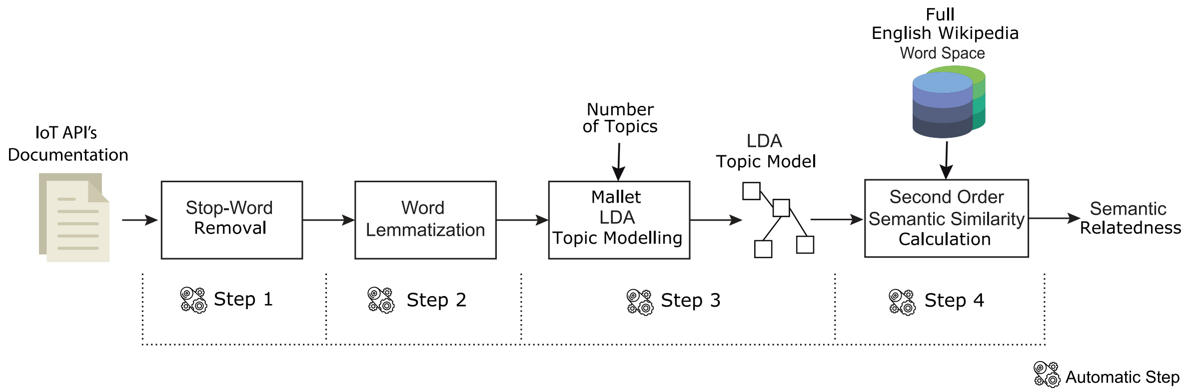

Fig. 4. The semantic analysis strategy of the SARAv2 approach.

contextual) similarity between identifiers. The distributional second-order similarity metric is useful for us because the nodes (*i.e.*, resource identifiers) might slightly differ from their actual API documentation syntactically and semantically. Two nodes (or words) can be seen as distributionally similar when they have co-occurring words in common, *i.e.*, common words as neighbours. We rely on DISCO (Kolb, 2008) library to compute the distributional similarity between nodes in a URI.

Table 2 shows similarity values for two URIs from Google Nest API: (1) `developer-api.nest.com/devices/thermostats/device_id/locale` and (2) `developer-api.nest.com/structures/structure_id/co_alarm_state`.

These values are computed based on the topic models and the distributional second-order similarity metric. In other words, if we want to compare the context of a pair of nodes in a URI, we compute the second-order semantic similarity between them with the top 15 words in each topic from the obtained topic model. Then, we decide the topic to which a node belongs to based on the similarity value, *i.e.*, a node fits a topic if the average second-order semantic similarity value is greater than the threshold 0.3. Also, for a pair of nodes, if the intersection of topics to which the nodes belong is *null* (*i.e.*, no common topic), then, the URI is regarded as an instance of *Contextless Resource Names* linguistic antipattern. In contrast, if each pair of nodes in a URI belongs to one or more common topic(s), we report the URI as an instance of *Contextual Resource Names* linguistic pattern.

For the first URI, the base form of each node (*i.e.*, *device*, *thermostat*, and *locale*) appears in Topic 3, except the node *locale*. Moreover, the average similarity value for all the nodes against Topic 1 is 0.4077, against Topic 2 is 0.3655, and against Topic 3 is 1.4875, which means the first URI is more similar to Topic 3 with a higher similarity value of 1.4875. The average similarity in Table 2 was computed by taking the maximum similarity value for each node in the URI against all the words in a topic, and then average them. For example, the maximum similarity values for the nodes in the first URI are 0.7259, 0.3378, and 0.1595 for Topic 1, averaging 0.4077, which is greater than the threshold of 0.3. The average similarity values for Topic 2 and Topic 3 are 0.3655 and 1.4875, respectively. Ergo, we identify the first URI as *Contextual Resource Names* linguistic pattern. For the second Google Nest URI, all the nodes *structure*, *alarm*, and *state* appear in Topic 3 in their base form except the node *state*. Similar to the first URI, the second URI also more fit Topic 3 with an average similarity value of 1.3576 (see Table 2). We identify the second URI as *Contextual Resource Names* linguistic pattern because all the nodes are semantically related.

3.3.3. Applying detection algorithms

In this work, we use SARAv2, which extends the SARA approach for the semantic analyses of REST URIs and APIs documentations used in Palma et al. (2015, 2017) by adding three new patterns and antipatterns. The extension includes the implementation of the new detection algorithms for the three newly defined patterns and antipatterns.

Both SARA and SARAv2 use and extend the SOFA framework proposed and developed by Moha et al. (2012) to automatically execute the detection heuristics in the form of detection algorithms on the URIs. SARA and SARAv2 extend SOFA by enabling the use of LDA models and Second Order Semantic Similarity as heuristics in the detection algorithms that analyse the URIs and their documentation. For example, the detection code, as shown in Listing 1, is implemented and executed inside the SOFA framework. The detection results are then exported to a text file.

3.4. Operationalising REST-Ling tool

REST-Ling is a web application that automates the detection of linguistic patterns and antipatterns. REST-Ling aims to help software engineers analyse their APIs and detect linguistic patterns and antipatterns. The tool can present various visual representations for patterns and antipatterns detected in a particular API. Moreover, it shows generic information on the types of linguistic patterns and antipatterns detected, and pinpoints the rationale behind their detection. Our REST-Ling tool supports:

- *The addition of APIs and URIs*: REST-Ling allows the user to add one or more URIs manually or by uploading a JSON file. The JSON file can contain multiple APIs with multiple URIs each;
- *The selection of patterns and antipatterns*: REST-Ling allows the user to select the patterns and antipatterns (both design and linguistic) to be detected. The analysis process is done asynchronously for all the patterns and antipatterns;
- *A detailed view of the detection results*: The tool provides answers to what, why, and where the design and linguistic antipatterns occur. This allows the user to have a better insight into the API quality by checking what type of antipatterns an API has;
- *A graphical representation of the detection results*: The tool provides a graphical representation of the detection results of the patterns and antipatterns using pie and bar charts;
- *A topic model creation feature for the linguistic analysis*: When it comes to detecting, for example, *Contextless Resource Names* linguistic antipattern, the tool provides functionalities to import and add acronyms and stop words to create the topic model required for the detection.

Example Use: To use the REST-Ling, engineers require a JSON file that contains a list of URIs from an API. Each URI should have a name, method, and description. Users can upload the JSON file on the `Collections` page to add the APIs to be analysed.

Table 2
An example analysis of two Google Nest URIs.

	Topic words	/devices/thermostats/device_id/locale			/structures/structure_id/co_alarm_state		
		Device	*Thermostat*	*Locale*	*Structure*	*Alarm*	*State*
Topic 1	eco	0.0528	0.0130	0.0275	0.0207	0.0146	**0.0064**
	record	0.0068	0.0000	0.0000	0.0000	0.0129	0.0000
	estimate	0.0611	0.0192	0.0241	0.0928	0.0239	0.0000
	lock	0.2254	**0.3378**	0.0055	0.1275	0.1955	0.0000
	adjust	0.2026	0.2575	0.0170	0.0721	0.1936	0.0000
	format	0.5311	0.0922	0.0451	0.1478	0.0990	0.0000
	related	0.0349	0.0129	0.0000	0.0327	0.0232	0.0000
	json	0.1962	0.0182	0.0253	0.0702	0.0313	0.0000
	call	0.0656	0.0081	0.0058	0.0648	0.0915	0.0000
	sign	0.0000	0.0000	0.0057	0.0000	0.0060	0.0058
	home	0.0000	0.0000	0.0703	0.0817	0.0000	0.0062
	sound	0.1645	0.0919	0.1588	0.1853	0.2277	0.0000
	bandwidth	**0.7259**	0.2130	0.0317	0.0944	**0.2545**	0.0000
	low	0.0517	0.0958	0.0113	0.0521	0.0636	0.0000
	image	0.3504	0.0788	**0.1595**	**0.5273**	0.1274	0.0000
	Average		**0.4077**			**0.2627**	
Topic 2	smoke	0.0054	0.1065	0.0058	0.0059	0.0609	0.0000
	sound	0.1645	0.0919	**0.1588**	0.1853	0.2277	0.0000
	snapshot	0.3066	0.0178	0.0395	0.0732	0.0863	0.0000
	status	0.0588	0.0135	0.1419	0.2592	0.0188	0.0291
	change	0.0976	0.0603	0.0805	0.3649	0.0852	0.0000
	list	0.0348	0.0069	0.0741	0.1001	0.0054	**0.1672**
	display	**0.7431**	0.1763	0.0842	0.2419	0.1894	0.0000
	nest	0.0054	0.0000	0.0495	0.0118	0.0000	0.0061
	expire	0.0442	0.0000	0.0060	0.0055	0.0072	0.0364
	home	0.0000	0.0000	0.0703	0.0817	0.0000	0.0062
	detect	0.3562	0.1291	0.0000	0.1933	**0.3026**	0.0000
	subscription	0.2463	0.0113	0.0059	0.0180	0.0687	0.0000
	field	0.1114	0.0193	0.1001	0.1900	0.0117	0.0136
	live	0.0000	0.0000	0.0836	0.0260	0.0062	0.0069
	motion	0.3378	**0.1946**	0.0657	**0.4712**	0.2228	0.0000
	Average		**0.3655**			**0.3137**	
Topic 3	device	**2.0000**	0.4916	0.0242	0.2111	0.4377	0.0000
	structure	0.2111	0.0569	0.1645	**2.0000**	0.0210	0.0000
	thermostat	0.4916	**2.0000**	0.0057	0.0569	0.3944	0.0000
	event	0.0175	0.0118	0.1557	0.0507	0.0152	0.0000
	nest	0.0054	0.0000	0.0495	0.0118	0.0000	0.0061
	camera	1.0680	0.3072	0.0177	0.1089	0.4096	0.0000
	url	0.2685	0.0117	0.0177	0.0577	0.0766	0.0000
	display	0.7431	0.1763	0.0842	0.2419	0.1894	0.0000
	temperature	0.0987	0.2084	0.0645	0.1731	0.0681	0.0000
	require	0.4377	0.3944	0.0000	0.0210	0.1611	0.0000
	alarm	0.3914	0.1334	0.0167	0.1543	**2.0000**	0.0140
	hvac	0.3133	0.8992	0.0000	0.0351	0.2735	0.0000
	aware	0.0060	0.0000	0.0000	0.0116	0.1789	0.0000
	zone	0.0729	0.0402	**0.4626**	0.2317	0.0195	**0.0727**
	activity	0.1565	0.0371	0.2037	0.3848	0.0912	0.0053
	Average		**1.4875**			**1.3576**	

Once the file is uploaded, the user can go into each `Collection` and start the analysis by clicking on the `Analyse` button and checking the detected patterns and antipatterns in the collection view, as shown in Fig. 5. The REST-Ling tool can be accessed on https://rest-ling.com. To use the tool, provide `admin` both as the username and password. A demo of the tool is provided on YouTube.[7] The tool is built to be freely used by anyone who aims to improve their APIs' design quality. The REST-Ling tool might be of interest to academics aiming to perform further research on API quality. Practitioners can also use the tool to assess the quality of their APIs.

4. Experiments and results

This section reports on two empirical studies using the SARAv2 approach. The first study in Section 4.2 aims at performing the qualitative analysis of IoT APIs utilising the REST-Ling tool. In the second study, in Section 4.3, we assess the effectiveness of the REST-Ling tool by validating the accuracy of the detection heuristics and the efficiency of the detection algorithms that are part of the underlying SOFA framework (Moha et al., 2012). In the following sections, we provide the details of the study results.

4.1. Subjects and objects

In these two empirical studies, we consider nine linguistic antipatterns and their corresponding patterns as discussed in Section 2. As our objects, we collected a list of more than 700 Web APIs from `programmableweb.com` of 73 types including 'Big Data', 'Cloud', 'Database-as-a-Service', 'Infrastructure-as-a-Service', 'Internet of Things', and 'Platform-as-a-Service'. From that list, we filtered only APIs related to the 'Internet of Things' and finally chose 19 IoT APIs that have well-organised API documentation. We manually extracted the URIs, their documentation, and underlying HTTP methods. We collected and analysed a set of 1102 URIs from the 19 IoT APIs. Table 3 lists the 19 IoT APIs

7 https://www.youtube.com/watch?v=pSdll8hOFjY.

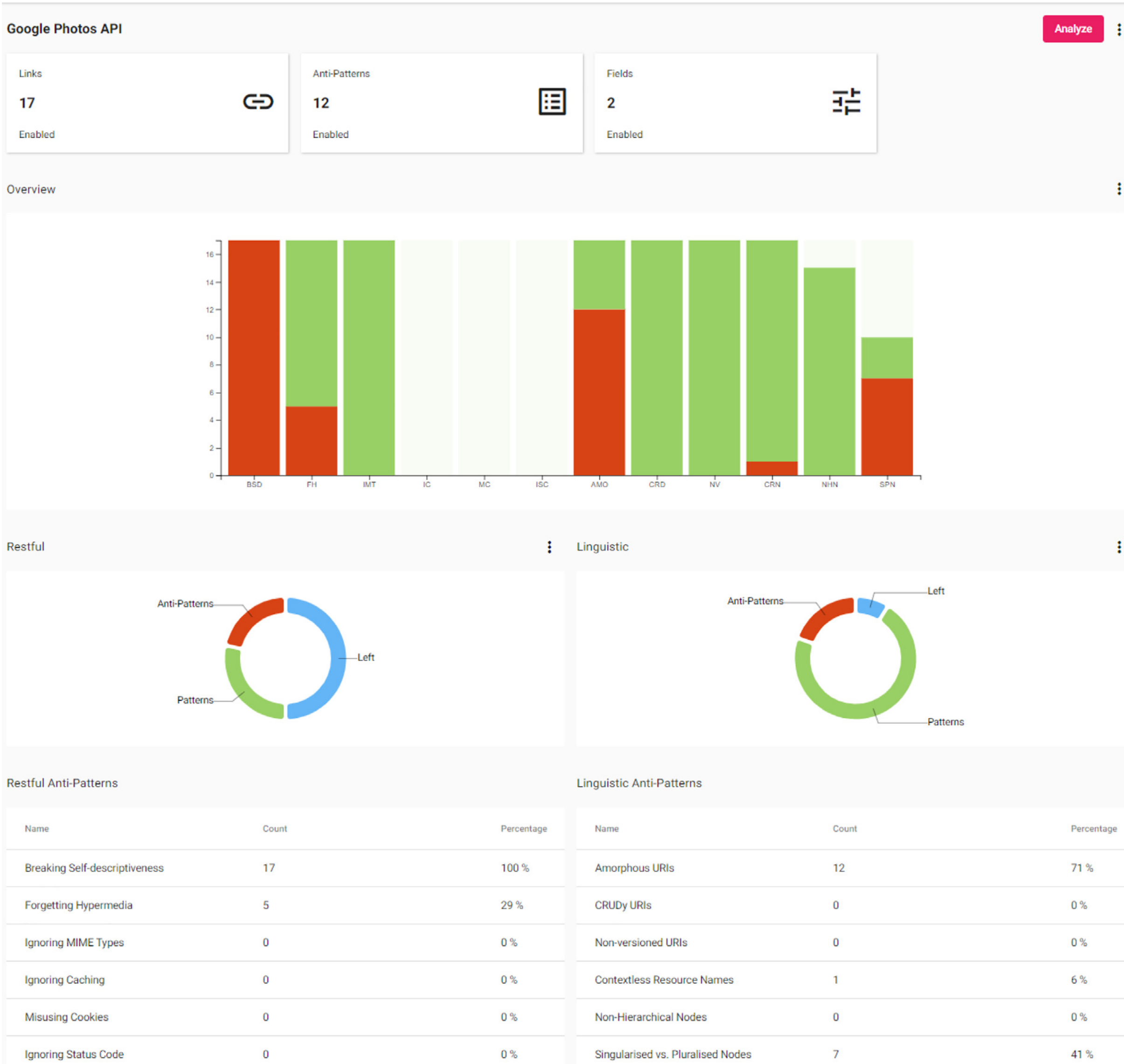

Fig. 5. Detection of patterns and antipatterns in an API.

and their online documentation that we analysed. We then apply detection heuristics of nine patterns and antipatterns as defined in Section 2 on the URIs to perform syntactic and semantic analyses. For all detection, we rely on the SOFA framework (Moha et al., 2012).

We inspected the documentation for the APIs to assess the support for MQTT. We found that 11 of them do not support MQTT in any way. We also found that in five of those that explicitly mention MQTT, only a subset of the full API functionality is supported, or the documentation is lacking. This leaves three APIs that claim full support for the MQTT protocol. This supports our claim that REST is the dominating style for developing cloud-based IoT applications.

4.2. Qualitative analysis of IoT APIs

This section provides an overview of the detection results. It should be noted that the detection is performed on the IoT APIs that had well-organised and self-contained documentation. Fig. 6 shows the detection summary of nine linguistic patterns and antipatterns on 19 IoT APIs. In Fig. 6, columns represent patterns and antipatterns, rows represent IoT APIs with the heights of the mosaics correspond to the count of URIs analysed for each API and the colours of the mosaic correspond to *white* as *pattern* detection and *black* as *antipattern* detection.

In Fig. 6, the most frequent linguistic patterns are: (a) Tidy URI, (b) Verbless URI, (c) Hierarchical Resource Names, and (d) Standard URI. More precisely, (a) almost all of the APIs (*i.e.*, 16 out of 19 analysed IoT APIs) had well-designed URIs in terms of lexical quality; (b) the majority of the analysed IoT APIs (*i.e.*, 12 out of 19 analysed IoT APIs) did not include any CRUDy (Create, Read, Update, Delete, and any of their synonyms) terms or the nodes in their URIs; (c) URI nodes are well-structured in a hierarchical fashion; and (d) APIs do not tend to include special and non-English characters in their URI design. In contrast, the most frequent antipatterns are: (i) Pluralised Nodes, (ii) Non-pertinent Documentation, and (iii) Unversioned URI. In particular, (i) for the PUT/DELETE requests, the last node of the URI should be singular, and for the POST requests, the last node should be plural, however, this was not always the case for IoT APIs; (ii) the documentation was not properly aligned with the URIs; and (iii) most of the IoT API providers did not use version information within URIs, which may hinder APIs maintainability. These

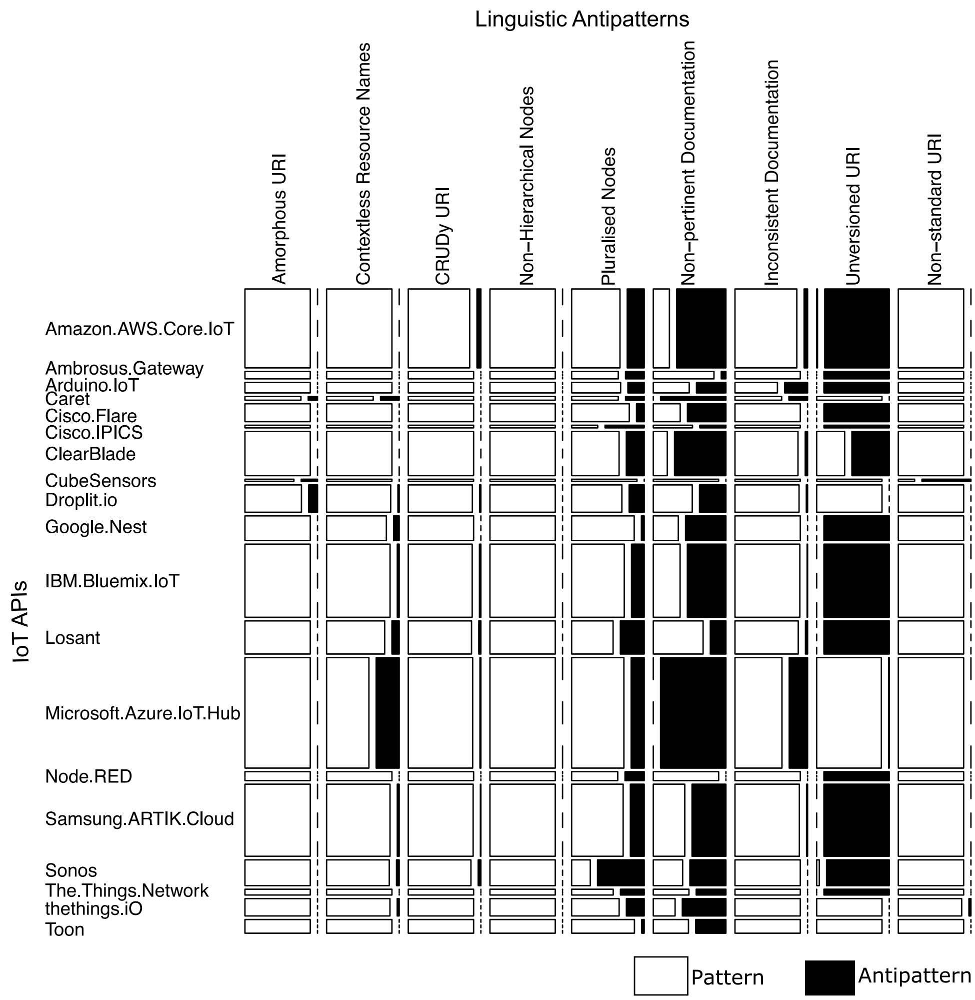

Fig. 6. Nine linguistic patterns and antipatterns detected in 19 IoT APIs using the REST-Ling tool. Columns represent patterns and antipatterns, rows represent IoT APIs with the heights of the mosaics correspond to the count of URIs analysed for each API.

Table 3
List of 19 analysed IoT APIs and their online documentations.

IoT APIs and online documentation	#URIs tested
Amazon AWS IoT Core	150
Ambrosus Gateway	14
Arduino IoT Cloud API	20
Caret	7
Cisco Flare	34
Cisco IPICS	5
ClearBlade	84
CubeSensors	4
Droplit.io	52
Google Nest	47
IBM Watson IoT	139
Losant	63
Microsoft Azure	210
Node-RED	17
Samsung ARTIK	137
Sonos	49
The Things Network	11
thethings.iO	33
Toon	26
Total	1102

conclusions are based on the detection results obtained using the REST-Ling tool.

Below, we briefly discuss the detection of some of the most and least common linguistic antipatterns.

CRUDy URI: The URI `/v0/api/auth/shortcode/create` from Droplit.io was detected as ✖*CRUDy URI*. The POST method was used to do that. Even without adding the 'create' node at the end of the URI, using the POST method, it was already understood that the goal was to create a shortcode that will be used for authentication, as stated in its documentation. The URL `/bulk/devices/remove` in IBM Watson IoT had a similar issue where it used the POST method to delete multiple devices. However, these poor practices of introducing CRUDy terms (or their synonyms) are highly discouraged in REST since there are a number of action-oriented HTTP methods available. The URI designers will simply combine an appropriate HTTP method from those with their URIs and perform diverse resource- or things-oriented tasks. The Samsung ARTIK also had similar issue, *e.g.*, a URI `/trials/sessions/search` is found with the 'search' node at the end.

Inconsistent Documentation: The ✖*Inconsistent Documentation* refers to the case where the HTTP method applies with a URI that has an opposite documentation, *i.e.*, the HTTP method does not do what it says. This is similar to *Method Signature and Comment are Opposite* linguistic antipattern in object-oriented programming (Arnaoudova et al., 2016, 2013). For example, from the Droplit.io API, there is a URI `/v0/api/clients/` that was applied with a GET method, but has the documentation as '*Create a client. An account token or server token may...*'. Clearly, using a GET method to create a client is poor (or even wrong) practice in REST. Similar instances were found in IBM Watson IoT, *e.g.*, it uses the POST method with the URI `/bulk/devices/remove` and has the documentation as '*delete multiple devices, each request can contain a...*'.

Contextless Resources Names: One URI `/devices/thermostats/device_id/time_to_target_training` by Google Nest was detected as ✖*Contextless Resources Names*. We suspect that the URI was detected as an antipattern because the nodes {`devices, thermostats, time, target, training`} seem not to be related from the semantic point of view. When we built the topic model for Google Nest, we found that 'device' and 'thermostat' were present in our topic model (under the first topic cluster) and the other three node words 'time', 'target', and 'training' were not in the topic model at all, *i.e.*, they were not so important in the context of Google Nest, thus, they were considered irrelevant to the context. In our topic model, similar words or keywords that are highly related are grouped under the same topic cluster. Therefore, the REST-Ling tool identified the URI as *Contextless Resources Names* antipattern.

Non-pertinent Documentation: Among the 19 analysed IoT APIs, all the APIs except two (*i.e.*, CubeSensors and Node-RED) have instances of this antipattern. Thus, *Non-pertinent Documentation* is found to be the most common antipattern. Also, 65% of the analysed URIs (*i.e.*, 712 out of 1102) are involved in this antipattern. These findings suggest that majority of the APIs (*i.e.*, 17 out of 19 analysed IoT APIs) do not provide documentation cohesive to their URI design. Also, large vendors like Amazon AWS Core IoT, IBM Watson IoT, and Google Nest do not tend to provide high-quality documentation for their APIs. For example, 75% of the analysed URIs from Amazon AWS Core IoT, 59% of the analysed URIs from IBM Watson IoT, and 62% of the analysed URIs from Google Nest had *Non-pertinent Documentation* antipattern. However, Microsoft Azure had 100% of the analysed URIs provided good quality documentation, i.e., the URIs and their documentation are cohesive.

Unversioned URI: We also found a similar prevalence for *Unversioned URI* antipattern, i.e., 14 out of 19 analysed IoT APIs do not include version information as part of their URIs design. In the literature, as part of the best practices, practitioners suggested including version information within the URIs (Levin, 2017). This is because APIs evolve continuously, and, if not properly versioned, clients might break. In other words, versioned APIs facilitate easy maintenance both for API providers and client developers. However, it could be due to that APIs for IoT applications do not evolve frequently. Notably, some APIs are found to be in two different modes, *i.e.*, some URIs are versioned whereas others are not. This observation ought not to be generalised without further investigation. For example, Amazon AWS Core IoT is found to have both unversioned and versioned URIs (148 *vs.* 2) out of 150 analysed URIs. Same for Microsoft Azure, where we found 2 out of 210 analysed URIs had version info included in the URI design. Thus, there is a clear lack of standardised practice among the API providers.

Non-Standard URI: According to the definition of *Non-standard URI* antipattern in Section 2.9, URI design should not include nodes or resources with non-standard identification (*e.g.*, special or unknown characters, blank spaces, double hyphens, etc.), which hinders the reusability and understandability of the APIs. Our findings suggest that the majority of the IoT APIs, *i.e.*, 17 out of 19 analysed APIs, follow the standard URI design practices. From the CubeSensors API, three URIs are found: `/devices/[deviceid]`, `/devices/[deviceid]/current`, and `/devices/[deviceid]/span` that a had blank space as part of the URI design. Also, the thethings.iO API had this URI `/things/THING_TOKEN/resources/$MAGIC_RESOURCE` with a dollar sign ($) before the parameter, which is considered as an unknown character in the URI design. These practices of URI design make the URI non-standard and hinder the reusability and understandability of the APIs.

4.3. Effectiveness of the REST-Ling

This section answers our research questions in showing the effectiveness of the REST-Ling tool.

4.3.1. Research questions

We define four research questions related to the prevalence, accuracy, and to assess the usefulness and effectiveness of the REST-Ling tool that is developed based on the SARAv2 approach.

- RQ_1 Prevalence: *To what extent IoT APIs suffer from poor linguistic design quality, i.e., linguistic antipatterns?* With RQ_1, we want to investigate whether IoT APIs suffer from linguistic design quality (*i.e.*, linguistic antipatterns), and to what extent.
- RQ_2 Comparison: *To what extent APIs across domains suffer from poor linguistic design quality, i.e., linguistic antipatterns?* With RQ_2, we want to investigate whether and to what extent APIs for Web applications and cloud services are prone to linguistic antipatterns than the APIs for IoT applications.
- RQ_3 Accuracy: *What is the accuracy of REST-Ling on the detection of linguistic antipatterns?* With RQ_3, we want to investigate the accuracy of our defined detection heuristics implemented in the REST-Ling tool.
- RQ_4 Efficiency: *How does the REST-Ling perform in terms of average detection time for linguistic antipatterns?* With RQ_4, we want to study the detection performance of the REST-Ling tool in executing the detection heuristics implemented as part of the tool. We conjecture that an average detection time in the order of seconds is acceptable for each antipattern.

4.3.2. Validation process

We applied random sampling to choose 91 URIs out of 1102 URIs from 19 APIs to measure the overall accuracy of the REST-Ling tool. The total population size is 9918 (*i.e.*, 1102 URIs × 9 patterns). We aim for a 95% confidence level and a confidence interval of 10; thus, a sample size of 819 questions (*i.e.*, 91 URIs × 9 antipatterns) were selected to be validated manually. We involved three professionals to manually validate our detection findings. The professionals have knowledge on REST and were not part of the implementation and execution of the detection algorithms. Two of the professionals co-author this paper and have both industrial and academic experience, and the third is an industry expert with working knowledge on Web APIs. To avoid any potential conflicts of interest and to be fully transparent, during the experiments and analyses, the obtained detection results were not shared and discussed with any of the authors who later participated in the validation process. In this way, we ensured that the accuracy of the detection results was not affected, *i.e.*, the accuracy measurements were unbiased.

To facilitate the validation process, the textual descriptions of linguistics patterns and antipatterns, the URIs along with the HTTP method, and their documentations were provided. To set the oracle, we decided on the majority. That is, each detection instance is manually validated by three participants and the oracle is decided when at least two participants accept or reject an instance. The validation process with questions and responses is done online and available using Google Forms,[8]. As shown in Eq. (1) the *accuracy* measure was used to measure the performance of the REST-Ling tool. We also use the Matthews' correlation coefficient (MCC), as shown in Eq. (2), as an alternative measure unaffected by the unbalanced datasets issue.

8 https://forms.gle/EmGPoZbRwGXybFHY8.

Table 4
Detection results of the nine linguistic patterns and antipatterns in 19 IoT APIs.

IoT APIs	✖Amorphous URI	✔Tidy URI	✖Contextless Resource Names	✔Contextualised Resource Names	✖CRUDy URI	✔Verbless URI	✖Non-Hierarchical Nodes	✔Hierarchical Nodes	✖Pluralised Nodes	✔Singularised Nodes	✖Non-pertinent Documentation	✔Pertinent Documentation	✖Inconsistent Documentation	✔Consistent Documentation	✖Unversioned URI	✔URI Versioning	✖Non-standard URI	✔Standard URI
Amazon AWS Core IoT	0	150	0	150	9	141	0	150	39	111	113	37	8	142	148	2	0	150
Ambrosus Gateway	0	14	0	14	0	14	0	14	4	10	1	13	0	14	14	0	0	14
Arduino IoT	0	20	0	20	0	20	0	20	5	15	9	11	7	13	20	0	0	20
Caret	1	6	2	5	0	7	0	7	2	5	7	0	2	5	0	7	0	7
Cisco Flare	0	34	0	34	0	34	0	34	4	30	20	14	0	34	34	0	0	34
Cisco IPICS	0	5	0	5	0	5	0	5	3	2	2	3	0	5	5	0	0	5
ClearBlade	0	84	0	84	0	84	0	84	23	61	66	18	3	81	48	36	0	84
CubeSensors	1	3	0	4	0	4	0	4	0	4	0	4	0	4	0	4	3	1
Droplit.io	7	45	1	51	1	51	0	52	12	40	21	31	1	51	0	52	0	52
Google Nest	0	47	4	43	0	47	0	47	2	45	29	18	0	47	47	0	0	47
IBM Bluemix IoT	0	139	4	135	3	136	0	139	27	112	82	57	1	138	139	0	0	139
Losant	0	63	7	56	2	61	0	63	23	40	15	48	2	61	63	0	0	63
Microsoft Azure IoT Hub	0	210	74	136	3	207	0	210	42	168	210	0	59	151	2	208	0	210
Node-RED	0	17	0	17	0	17	0	17	5	12	0	17	0	17	17	0	0	17
Samsung ARTIK Cloud	0	137	4	133	1	136	0	137	29	108	71	66	2	135	137	0	0	137
Sonos	0	49	2	47	2	47	0	49	35	14	27	22	0	49	47	2	0	49
The Things Network	0	11	0	11	0	11	0	11	4	7	5	6	0	11	11	0	0	11
thethings.iO	0	33	1	32	0	33	0	33	9	24	22	11	0	33	0	33	1	32
Toon	0	26	0	26	0	26	0	26	1	25	12	14	0	26	0	26	0	26

The MCC is computed based on the contingency matrix. MCC generates a higher score if our REST-Ling tool as a binary classifier can correctly classify the majority of antipattern instances and the majority of the pattern instances. MCC has the values −1 and +1 for a perfect misclassification and perfect classification, respectively.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (2)$$

TP = True Positive; FP = False Positive; TN = True Negative; and FN = False Negative.

In the following sections, we answer the four research questions as stated in Section 4.3.1.

4.3.3. RQ_1 prevalence

Table 4 presents detection results for the nine pairs of linguistic patterns and antipatterns on 19 IoT APIs. In Table 4, the first column shows the linguistic patterns and antipatterns followed by the 19 IoT APIs. For each API and for each pattern and antipattern, the total number of occurrences is reported that are found as positives by our detection algorithms. The last column shows the total occurrences with percentage for each pattern and antipattern. The detailed analyses results for all the 1102 URIs from 19 IoT APIs are available online.[9]

To summarise the results in Table 4, Amazon AWS Core IoT, Arduino IoT, Caret, Cisco IPICS, and Sonos had the most number of antipatterns given the number of URIs tested for each of those APIs. In contrast, Google Nest, Node-RED, CubeSensors, thethings.iO, and Toon had the most number of patterns given the number of URIs tested for each those APIs. We make these observations by dividing the total instances of patterns or antipatterns by the total number of analysed URIs for each API.

As Table 4 suggests, all the analysed IoT APIs contain at least one of nine linguistic antipatterns. The set of five IoT APIs, *i.e.*, Droplit.io, IBM Watson IoT, Losant, Microsoft Azure IoT Hub, and Samsung ARTIK Cloud, is found to be involved in six different linguistic antipatterns. Although, CubeSensors, Node-RED, and Toon APIs are found to be involved in only two linguistic antipatterns.

Summary on RQ_1: Linguistic antipatterns are prevalent in IoT APIs. In the analysed IoT APIs we have detected some instances of poor design practices, being the most prevalent *Non-pertinent Documentation* or *Unversioned URIs*. We also observed the presence of good design practices, *i.e.*, linguistic patterns, which suggests that the developers are aware of the need for linguistic quality on their APIs.

4.3.4. RQ_2 comparison

APIs are used for various purposes and in various domains, for example, APIs for Web applications (Palma et al., 2017) and Cloud services (Petrillo et al., 2018b). In this research, we aim to find whether a certain linguistic antipattern (or pattern) is notably common across the domains or whether a certain domain is more prone to an antipattern (or pattern) compared to other domains. It is important to note that the methods relevant to the domains, i.e., SARA (Palma et al., 2017) and CloudLex (Petrillo et al., 2018b) are applied to different sets of APIs. Thus, a direct comparison among the methods is not possible. Instead, we only want to compare the prevalence of linguistic antipatterns in REST APIs in different domains.

As Table 5 shows, the APIs for Web applications have more ✖*Amorphous URI* (65%) than the IoT APIs (0.82%). In contrast, IoT APIs are more often tidy than the APIs from the other domains with 99.18% URIs are detected as ✔Tidy URI. A consistent detection is observed in Table 5 for the ✖*Contextless Resource Names* and its corresponding ✔*Contextualised Resource Names*, *i.e.*, the majority of the URIs in the three domains are designed with resource

9 https://doi.org/10.5281/zenodo.6393404.

Table 5
Comparison of the detection of linguistic antipatterns across domains.

Approach	SARA (Palma et al., 2017)		CloudLex (Petrillo et al., 2018b)		SARAv2	
Target domain	APIs for Web apps		APIs for Cloud services		APIs for IoT	
Number of APIs analysed	18		16		19	
Number of URIs tested	310		23,062		1102	
(Anti)Patterns/Detection	#Instances	%URIs	#Instances	%URIs	#Instances	%URIs
✖Amorphous URI	202	65%	–	–	9	0.82%
✔Tidy URI	108	35%	–	–	1093	99.18%
✖Contextless Resource Names	123	40%	10,595	23%	99	8.98%
✔Contextualised Resource Names	187	60%	12,467	77%	1003	91.02%
✖CRUDy URI	38	12%	–	–	21	1.91%
✔Verbless URI	272	88%	–	–	1081	98.09%
✖Inconsistent Documentation	–	–	–	–	85	7.71%
✔Consistent Documentation	–	–	–	–	1017	92.29%
✖Non-hierarchical Nodes	173	56%	–	–	0	0%
✔Hierarchical Nodes	0	0%	–	–	1102	100%
✖Non-pertinent Documentation	155[a]	28%	1339[b]	52%	712	64.61%
✔Pertinent Documentation	400[a]	72%	792[b]	48%	390	35.39%
✖Pluralised Nodes	14	5%	–	–	269	24.41%
✔Singularised Nodes	4	1%	–	–	833	75.59%
✖Unversioned URI	–	–	–	–	732	66.42%
✔Versioned URI	–	–	–	–	370	33.58%
✖Non-standard URI	–	–	–	–	4	0.36%
✔Standard URI	–	–	–	–	1098	99.64%

[a]Detection was done on additional set of URIs.
[b]Detection was done on a subset of the URIs.

names that are semantically aligned within the context of the URI design. Thus, 60% of URIs for Web applications, 77% of URIs for Cloud services, and 91.02% of URIs for IoT APIs are detected as ✔*Contextualised Resource Names*. Similarly, designers are well aware of not using verbs within the URI design, thus, 88% URIs for Web applications and 98.09% URIs from the IoT APIs are detected as ✔*Verbless URI*. To summarise, APIs for Web applications are mostly prone to ✖*Amorphous URI* and notably implement patterns like ✔*Verbless URI* and ✔*Pertinent Documentation*. On the other hand, IoT APIs suffer mostly with ✖*Non-pertinent Documentation* and ✖*Unversioned URI*, which suggests that IoT APIs are poorly documented and the URI designers do not tend to design the URIs with version info – a poor URI design practice. In contrast, we found that IoT APIs have very tidy URIs, *i.e.*, the ✔*Tidy URI*, and the nodes in the URIs are organised hierarchically, *i.e.*, the ✔*Hierarchical Nodes*.

More specifically, for Web APIs in Palma et al. (2017), for example, Facebook had a high number of ✖*Contextless Resource Names* and ✖*Non-hierarchical Nodes* due to its diverse and large set of resources. It is often difficult to find a best hierarchical order of URIs nodes or to find resources names best fit to a certain context. However, Twitter and YouTube, for example, did not suffer those antipatterns with comparatively lower number of resources than Facebook (Palma et al., 2017). In fact, on average, StackExchange had the most number of antipatterns, due to which ✖*Amorphous URI* and ✖*Non-hierarchical Nodes* seem very common, as reported by SARA (Palma et al., 2017).

On the contrary, relatively new IoT APIs are designed with more knowledge and experience from the literature of good design practices and guidelines on APIs design (Massé, 2012; Fredrich, 2012; Microsoft MSDN, 2006; Levin, 2017; Riggins, 2015; Keranen et al., 2017). This could be one reason the ✖*Amorphous URI* is found on very small scale. Also, the detection for ✖*Contextless Resource Names* and ✖*Non-hierarchical Nodes* in IoT APIs resulted in comparatively lower than in APIs for Web applications. The major IoT APIs vendors including Amazon, Google, IBM, and Microsoft are well aware of designing quality URIs both syntactic (*e.g.*, ✖*Amorphous URI*) and semantic (*e.g.*, ✖*Contextless Resource Names*, ✖*Non-hierarchical Nodes*, or ✖*Inconsistent Documentation*) viewpoints.

Overall, on average, 34% of the URIs from the APIs for Web applications are detected having linguistic antipatterns by SARA (Palma et al., 2017). In contrast, only 17% of the URIs from the APIs for IoT devices are detected as antipatterns by the REST-Ling tool. Also, for linguistic patterns, the REST-Ling tool found 73% URIs are well-designed compared to 42% URIs of APIs for Web applications. This suggests that IoT APIs are comparatively more well-designed than the APIs for Web applications like Facebook, YouTube, or Instagram (Palma et al., 2017). This could be because the APIs specific to Web applications deal with a plethora of resources types and representations, compared to the APIs in the IoT domain, where devices mainly deal with device data and transmission from/to the servers and peer devices. Thus, APIs for Web applications pose a higher challenge in designing high-quality URIs than the IoT APIs, *i.e.*, APIs for Web applications are more prone to linguistic antipatterns.

Summary on RQ$_2$: We found that resource URIs are structurally and contextually well-designed in APIs for IoT applications than for Web applications. Although the APIs for cloud services are not studied to a large number, the analysis of resource context (*Contextless vs. Contextualised Resource Names*) and cohesive documentation (*Pertinent vs. Non-pertinent Documentation*) suggests that APIs for cloud services exhibit similar design quality found in APIs for Web applications. In fact, **APIs for IoT applications appear to have a better design (structural and contextual) except that the APIs for IoT applications are poorly and, in many cases, briefly documented**.

4.3.5. *RQ$_3$ accuracy*

Table 6 shows the detection accuracy for nine linguistic antipatterns. In Table 6, on a subset of 91 URIs from 19 IoT APIs, we obtained an average accuracy of 81%. The accuracy is also heavily dependent on how engineers (in our case, the three professionals) understand and interpret a phrase or word based on their experience and knowledge. For example, in Validation 1, an

Table 6
REST-Ling validation results to compute overall precision and recall.

Linguistic antipatterns	P	N	TP	FP	FN	TN	Accuracy	MCC
✖*Amorphous URI*	4	87	1	3	27	60	67%	−0.03
✖*Contextless Resource Names*	11	80	6	5	15	65	78%	0.28
✖*CRUDy URI*	3	88	3	0	3	85	97%	0.69
✖*Non-hierarchical Nodes*	0	91	0	0	14	77	85%	*n/a*
✖*Pluralised Nodes*	31	60	25	6	5	55	88%	0.73
✖*Non-Pertinent Documentation*	58	33	8	50	4	29	41%	0.02
✖*Unversioned URIs*	67	24	67	0	0	24	100%	1.00
✖*Inconsistent Documentation*	26	65	19	7	8	57	84%	0.60
✖*Non-standard URI*	2	89	2	0	10	79	89%	0.38
Total	**202**	**617**	**131**	**71**	**86**	**531**		
Average							**81%**	**0.46**

Listing 2: Code snippet for measuring the detection time.

```
1 /*Detection of Contextless Resource Names Antipattern*/
2 long startTime = System.currentTimeMillis();
3 detectContextlessResourceNames();
4 long endTime = System.currentTimeMillis();
5 System.out.println("Total detection time (sec): " + (double)
      (endTime - startTime)/1000);
```

instance from Losant with the URI /applications/APPLICATION_ID/devices/DEVICE_ID/commandStream and the documentation[10] where the REST-Ling tool detects it as *Non-pertinent Documentation* antipattern, but the majority of the professionals (*i.e.*, two out of three) considered the URI and its documentation cohesive, thus, decided as *Pertinent Documentation* pattern. In another example, an instance from Cisco Flare with the URI /environments/{environment_id}/zones/{zone_id}/things/{thing_id}/data and with the documentation[11] where the REST-Ling tool approach detects it as *Pertinent Documentation* pattern, however, two out of three professionals did not see this URI and its documentation cohesive, and identified it as *Non-pertinent Documentation* antipattern. Instances similar to the above examples may lead to lower accuracy.

> **Summary on RQ$_3$:** The manual validation suggests that the REST-Ling tool has an overall **average accuracy of more than 80%**, with an average MCC of 0.46.

4.3.6. RQ$_4$ efficiency

We performed the experiments on an Intel Dual Core at 3.30 GHz with 4 GB of RAM. For the detection of linguistic patterns and antipatterns in IoT APIs, the reported detection times include: (1) the time to apply and run the detection algorithms implemented in Java on the URIs and (2) the time to export the results. Listing 2 shows the code snippet we use for measuring the detection time for each API and for each linguistic antipattern where we record the time before and after running the detection (lines 4 and 6). We then take the difference and get the values in seconds (line 7). Table 7 shows the detection time for each API (rows) and each antipattern (columns).

Regardless of the APIs, for each antipattern, we observed a consistent detection time, *i.e.*, when the number of tested URIs for

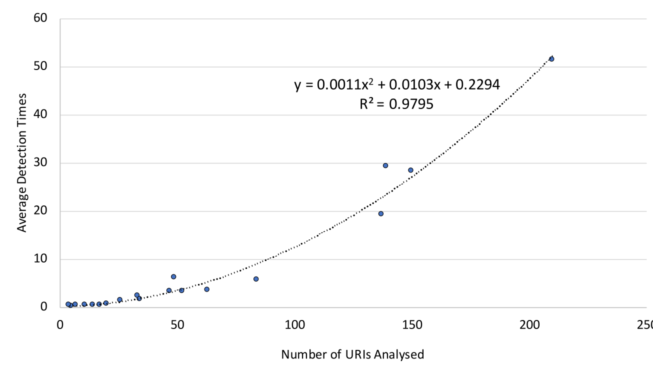

Fig. 7. Plot of the average detection times against the URIs analysed for the APIs.

an API is low, the detection time is lower and the detection time increases polynomially when the number of tested URIs increases. The estimated growth function is $y = 0.0011x^2 + 0.0103x + 0.2294$ where x is the number of URIs to test, and y is the total detection time for an antipattern. This growth function is particularly applicable for the *Contextless Resource Names* antipattern where we needed to perform a significant amount of pairwise comparisons for the nodes in the URIs. Also, the detection time is considerably lower when the detection does not require exhaustive comparisons or only requires syntactic checking, *e.g.*, *Amorphous URI*, *CRUDy URI*, *Unversioned URI*, *Non-standard URI*, and so on. For example, the detection times for *Amorphous URI* antipattern are between 0.001 and 0.089 s, the detection times for *Pluralised Nodes* antipattern are between 0.002 and 0.021 s. However, the detection times for *Contextless Resource Names* antipattern are between 0.459 (for CISCO IPICS with only five tested URIs) and 351.087 s (for Microsoft Azure with 210 tested URIs). And, in this case, the detection times depend on the exhaustive semantic comparisons among the nodes in the URIs.

As shown in Table 7, the global average detection time of each antipattern for all IoT APIs is 8.396 s. Our detection algorithms for linguistic antipatterns have a polynomial complexity of $\mathcal{O}(n^k)$. The average detection time can be expressed as a polynomial function as shown in Fig. 7.

> **Summary on RQ$_4$ Efficiency:** In concern to efficiency of REST-Ling, we want to achieve an average detection time for each linguistic antipattern and for each IoT API in the order of seconds. Regardless of the number of URIs tested for each API, **the REST-Ling had an average detection time of 8.396 s**. Moreover, the total detection time for nine antipatterns and nine patterns on 19 APIs with 1102 URIs was 1435.727 s.

[10] Attach to a real time stream of command messages to this device using SSE.

[11] Get thing data. Gets all data values for a thing by making a REST call. You can also get the data for a thing using the getData Socket.IO call.

Table 7
Detection times (in seconds) of the nine linguistic patterns and antipatterns in 19 IoT APIs.

IoT APIs	Amorphous URI	CRUDy URI	Non-hierarchical Nodes	Pluralised Nodes	Unversioned URI	Non-pertinent Doc.	Contextless Resource	Inconsistent Doc.	Non-standard URI	Total	Average
Amazon AWS IoT	0.003	0.033	1.307	0.009	0.004	59.462	193.442	0.010	0.002	254.272	28.252
Amrosus Gateway	0.002	0.005	1.356	0.005	0.002	2.092	0.777	0.012	0.001	4.252	0.472
Arduino IoT	0.002	0.009	1.278	0.003	0.002	2.833	3.103	0.010	0.001	7.241	0.805
Caret	0.001	0.003	1.297	0.021	0.001	1.142	1.544	0.013	0.001	4.023	0.447
Cisco Flare	0.004	0.014	1.292	0.005	0.004	10.288	4.405	0.015	0.001	16.028	1.781
Cisco IPICS	0.002	0.005	1.342	0.004	0.001	1.037	0.459	0.010	0.001	2.861	0.318
ClearBlade	0.003	0.015	1.306	0.010	0.006	11.681	39.152	0.014	0.001	52.188	5.799
CubeSensors	0.001	0.002	1.317	0.002	0.001	1.667	0.610	0.014	0.001	3.615	0.402
Droplit.io	0.002	0.009	1.331	0.008	0.004	14.200	14.264	0.011	0.001	29.830	3.314
Google Nest	0.047	0.008	1.166	0.006	0.005	19.223	10.057	0.010	0.001	30.523	3.391
IBM Watson	0.004	0.060	1.275	0.012	0.007	75.337	187.547	0.010	0.040	264.292	29.366
Losant	0.003	0.029	1.244	0.008	0.003	12.385	18.489	0.014	0.002	32.177	3.575
Microsoft Azure	0.011	0.052	1.302	0.019	0.041	109.248	351.087	0.012	0.006	461.778	51.309
Node-RED	0.009	0.003	1.347	0.004	0.002	1.703	1.564	0.013	0.001	4.646	0.516
Samsung ART	0.046	0.018	1.250	0.012	0.005	47.180	125.294	0.011	0.002	173.818	19.313
Sonos	0.089	0.010	1.214	0.009	0.005	31.766	21.943	0.012	0.001	55.049	6.117
The Things Net	0.016	0.004	1.227	0.003	0.001	2.875	1.400	0.009	0.001	5.536	0.615
thethings.io	0.044	0.011	1.280	0.004	0.007	13.009	7.154	0.010	0.001	21.520	2.391
Toon	0.040	0.088	1.197	0.005	0.004	8.006	2.727	0.010	0.001	12.078	1.342
Total	0.329	0.378	24.328	0.149	0.105	425.134	985.018	0.220	0.066	1435.727	
Average	0.017	0.020	1.280	0.008	0.006	22.375	51.843	0.012	0.003		**8.8396**

4.4. Threats to validity

Our findings may not be generalised to all IoT APIs. However, to minimise the threat to the external validity of our results, we performed experiments on a set of 1102 URIs from 19 IoT APIs. To minimise the threat to the internal validity, we not only used WordNet (Fellbaum, 1998) for lexical analyses of URIs, but we also relied on a technique based on the LDA topic modelling to properly capture the context and use the DISCO second-order similarity metric to measure the similarity between the nodes in URIs. But, the outcome of the detection may vary depending on the way the detection heuristics of linguistic patterns and antipatterns are applied because the software engineers may have their own understanding and experience on IoT and on the linguistic patterns and antipatterns. Moreover, in the manual validation, only 91 of the 1102 URIs were validated manually, which may be not representative. However, to minimise the threat, we aimed for a 95% confidence level and a confidence interval of 10, thus, we ended up validating 91 URIs and nine linguistic antipatterns, *i.e.*, 819 questions.

In this study, the detection of linguistic antipatterns is performed on the IoT APIs that had well-organised and self-contained documentation. Thus, the detection on IoT APIs with no or very minimal documentation could yield different results. However, we minimised the threats to the construct validity by selecting a set of IoT APIs that are well-documented. Moreover, we tried to minimise the threat to the construct validity by defining detection heuristics after a thorough review of definitions of linguistic patterns and antipatterns. Three professionals were involved in the validation process, and we decided the oracle based on the majority (*i.e.*, when two participants agreed out of the three who participated in the manual validation). The average degree of agreement among the professionals for the manual validation was 0.83. Thus, an average agreement of 0.83 for all antipatterns helps to minimise the threat to the construct validity, *i.e.*, our validation results and accuracy are reliable. The REST-Ling tool currently supports the detection of nine linguistic patterns and nine linguistic antipatterns. The tool has a detection accuracy of more than 80%. The accuracy of the tool is confirmed via manual validation of the detection outcomes.

In fact, we cannot claim the list of linguistic patterns and antipatterns is complete. Therefore, when talking about linguistic quality of IoT APIs, we refer to the set of linguistic patterns and antipatterns in our study. Nevertheless, to the best of our knowledge, our approach and the empirical study performed on REST IoT APIs is the most comprehensive analysis so far.

To minimise the threats to validity, *i.e.*, to increase the reliability and replicability, we have put all the details of this study online.[12]

5. Related work

Analysis techniques to detect linguistic patterns and antipatterns have been previously applied to Object-Oriented (OO) systems' source code *e.g.*, Arnaoudova et al. (2016) and Abebe et al. (2009) to assess their linguistic quality (Section 5.1). In addition, several Natural Language Processing (NLP) techniques have been also applied to studying the linguistic quality of APIs and their documentation *e.g.*, Massé (2012), Petrillo et al. (2016, 2018a), OMA (2012), Petrillo et al. (2018b), Haupt et al. (2017), Hausenblas (2011) and Brabra et al. (2016), as discussed in Section 5.2.

5.1. Syntactic and semantic analysis of source code

Abebe et al. (2009) presented a set of lexicon bad smells in OO code and a tool-suite using semantic analysis techniques to detect them. The authors aimed at improving the linguistic quality of OO source code because high quality and self-descriptive source code comments are useful in developing highly maintainable systems. Khamis et al. (2010) proposed the `JavadocMiner` approach for assessing the quality of in-line documentation relying on heuristics both in terms of language quality and consistency between source code and comments.

12 https://doi.org/10.5281/zenodo.6393404.

Semantic analyses have also been applied to Web services design analysis and development (Mateos et al., 2015; Rodriguez et al., 2010). Rodriguez et al. (2010) presented a study on poor linguistic practices identified on a set of WSDL (Web Service Definition Language) descriptions and provided a catalogue of Web services discoverability antipatterns. These antipatterns focus on the comments, elements names, or types used for representing the data models in WSDL documents. Also, Mateos et al. (2015) presented a tool to detect a subset of the antipatterns proposed in Rodriguez et al. (2010).

Other researchers also used semantic analyses in different phases of the software development life-cycle (Arnaoudova et al., 2014; Lu et al., 2015; Rahman and Chanchal, 2015). For example, Lu et al. (2015) proposed an approach to improve code searches by identifying relevant synonyms using the WordNet English lexical database (Fellbaum, 1998). Arnaoudova et al. (2014) performed a study on identifiers renaming in OO systems. Finally, Rahman and Chanchal (2015) presented an approach to automatically suggest relevant search terms based on the textual descriptions of change tasks in software. These approaches are tailored to OO identifiers and their consistencies with comments (Arnaoudova et al., 2016; Abebe et al., 2009) or to traditional SOAP-based Web services interfaces (Mateos et al., 2015; Rodriguez et al., 2010). Therefore, they cannot be applied to IoT APIs due to the peculiarities of their development life-cycle and their consumption nature.

Arnaoudova et al. (2016) presented the definition of *linguistic antipatterns* and defined 17 linguistic antipatterns in OO programming (*i.e.*, recurring poor practices related to inconsistencies among the naming, documentation, and implementation of a software entity), and implemented their detection algorithms. They searched for the differences between the identifiers used for software entities (*e.g.*, method names and return types) and their implementation and–or documentation. For example, one antipattern is called *"Is" returns more than a Boolean*, which analyses the name of a method starting with "Is" and checks if the method returns a boolean (Arnaoudova et al., 2016).

Machine learning techniques are also applied in predicting poor design, *i.e.*, antipatterns. For example, Fakhoury et al. (2018) performed a comparative study to explore how conventional machine learning classifiers perform compared to the deep learning methods in predicting linguistic antipatterns in object-oriented (OO) source code. Aghajani et al. (2018) conducted a large-scale study on more than 1.5k releases of 75 Maven libraries, 14k open-source Java projects based on those libraries, and more than 4k questions related to the libraries from Stack Overflow. More precisely, they studied if client developers are prone to introducing bugs when using APIs involved in linguistic antipatterns. Based on their statistical analysis, it is likely that linguistic antipatterns have an effect on introducing bugs (thus, triggering questions on Stack Overflow) with a probability of 29%. However, both these studies (Fakhoury et al., 2018; Aghajani et al., 2018) were conducted for linguistic antipatterns in OO source code.

5.2. Syntactic and semantic analysis of APIs and their documentation

APIs are the *de facto* standard used by software companies to design, develop, and offer their services through the Internet. Client developers must follow well-documented APIs to use the services and resources offered by those APIs properly. However, there are only a few standards and guidelines that guide API design and development (Massé, 2012; OMA, 2012).

Bertolino et al. (2009) modelled the SOAP Web service behaviour protocol, *i.e.*, how clients should behave while interacting with the service. They proposed the StrawBerry (Bertolino et al., 2009) method to automatically derive the Web service behaviour protocol from its WSDL interface.

As one of the first studies on APIs, Parrish (2010) did a subjective lexical comparison between two well-known APIs, *e.g.*, Facebook and Twitter. The author analysed, for example, the use of verbs and nouns in URIs naming and concluded that developers should rely on nouns instead of verbs while designing REST URIs.

Massé (2012) proposed an extensive list of REST API design principles, including the design of URIs, appropriate use of HTTP methods and status codes, meta-data design, and best practices for resource representations. The Open Mobile Alliance (OMA) provides guidelines for designing APIs exhibiting RESTfulness and for properly documenting the APIs, for example, not using verbs as resource identifiers or specifying API version within URIs. Several studies have also been performed in the domain of APIs automatically analyse their structure. For example, Haupt et al. (2017) presented a framework for structural analysis of APIs based on their documentations. They focused on the structural properties of APIs, and later, extended the study towards API governance (Haupt et al., 2018).

Panziera and Paoli (2013) 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. They evaluated their framework and reported the accuracy of identifying resources correctly with precision and recall of 72% and 77%, respectively.

Treude et al. (2015) developed a search-based approach for automatically extracting tasks (*i.e.*, a set of specific programming actions to be undertaken) from software documentation. They tried to minimise the gap between the information needs of the developers' and the documentation structure/content and, thus, assist developers in documentation navigation. Using the suggested approach, which utilises natural language processing techniques, they extracted more than 70% tasks from two large corpus of software documentation.

Some studies investigate and analyse services interfaces to measure their linguistic quality, in particular for SOAP Web services (Wei et al., 2015; Bertolino et al., 2009) and for APIs (Petrillo et al., 2016; Rodríguez et al., 2016). For example, Wei et al. (2015) presented a framework and algorithms to analyse service interfaces, the SOAP Web services, in particular. They targeted large and overloaded services with the goal to ease their integration and interoperability. The framework enabled to refactor large interfaces and was validated with real commercial logistic systems like FedEx.

Petrillo et al. (2016) provided a survey on REST literature and gathered 73 best practices in designing APIs to increase their understandability and reusability. They evaluated three well-known APIs from three Cloud providers, *i.e.*, Google Cloud Platform, OpenStack, and Open Cloud Computing Interface (OCCI), to evaluate their quality based on the identified best practices.

Rodríguez et al. (2016) analysed high-volume of REST HTTP traffic, *i.e.*, HTTP requests, to evaluate how well or bad developers implement APIs in practice. They compared the wellness with theoretical Web engineering principles and guidelines. The authors relied on heuristics and metrics to measure the implementation quality by means of antipatterns. Results showed a gap between theory and practice.

In our previous work (Palma et al., 2017), we proposed the SARA approach for automatically assessing the quality of APIs for Web applications through the detection of linguistic patterns and antipatterns. For the detection of linguistic patterns and antipatterns SARA relied on syntactic and semantic analysis of APIs. In another work (Petrillo et al., 2018b), we proposed CloudLex and studied the presence of linguistic patterns and antipatterns in 16 cloud computing APIs. The Cloud APIs tend to use heterogeneous

Table 8
Comparison with the relevant studies in the literature.

Study	Goal of the study	Target APIs	Analysis method
Parrish (2010)	Lexical analysis of 2 social network APIs	Facebook and Twitter	Identification of verbs and nouns
Wei et al. (2015)	Structural analysis of the service interfaces	272 operations from 13 cloud services related to Amazon, FedEx, etc.	Number of operations each service provides, average number (per operation) of input parameters, output parameters, business entities, etc.
Brabra et al. (2016)	Definition and detection of OCCI REST patterns and antipatterns	6 APIs for Cloud services	Detection of 28 OCCI REST antipatterns and their corresponding patterns
Petrillo et al. (2016)	Check APIs for conformance to 73 best practices	3 Cloud APIs (Google Cloud Platform, OpenStack, and OCCI)	For each Cloud API, manually checking the conformance to best practices at the service level
Rodríguez et al. (2016)	Check APIs for compliance or violation to 6 standardised practices	78 GB of HTTP traffic from Telecom Italia	For each action (Post, Get, Put, Delete, etc.) check for the conformance with standardised semantics, and also structural design of request URIs
Haupt et al. (2017)	Structural analysis of APIs	286 Swagger API description documents	Size of APIs in terms of resources, Number of POST/DELETE methods, read only resources, etc.
Palma et al. (2017)	Detection of linguistic antipatterns in APIs	18 APIs for Web applications	Heuristics-based detection for 6 linguistic antipatterns and their corresponding patterns
Petrillo et al. (2018a)	Linguistic quality assessment of Cloud Computing APIs	23,062 URIs from the 16 Cloud API providers	Detection of 2 linguistic antipatterns and their corresponding patterns
Brabra et al. (2019)	Detection of OCCI REST patterns and antipatterns	5 Cloud APIs including OCCI, COAPS, OpenNebula, Amazon S3, and Rackspace	Detection of 21 OCCI REST antipatterns and their corresponding patterns
SARAv2	Detection of linguistic antipatterns and patterns in APIs from IoT providers	19 APIs from 18 IoT providers	Heuristics-based detection for 9 linguistic antipatterns and their corresponding patterns

terms in their URI designs, and more than half of the URIs were not well-documented. CloudLex showed an average precision of 85% and a recall of 64%. In previous works, we also performed studies (Palma et al., 2015, 2017) that focused on the 'RESTful' aspect of Web APIs, for example, to see if the APIs follow basic REST design principles including (i) statelessness, (ii) cacheability, and (iii) interface uniformity.

In similar lines of research, working with OCCI patterns and antipatterns, Brabra et al. (2016, 2019) defined a set of patterns and antipatterns, inspired by the OCCI guidelines.[13] They performed an automatic detection of 28 OCCI REST patterns and antipatterns in Cloud APIs by invoking more than 300 operations.

5.3. State-of-the-art summary

The analysis of the aforementioned studies allow us to identify some limitations. More specifically, studies dedicated to the OO systems (Arnaoudova et al., 2016; Abebe et al., 2009) or SOAP-based Web services interfaces (Mateos et al., 2015; Rodriguez et al., 2010) are not applicable to APIs for IoT applications. Although, there are guidelines on API design (Massé, 2012), the semantic aspects of API design were considered in very few works (Haupt et al., 2017; Haupt et al., 2018). Some studies analysed the APIs or their documentation but did not assess the linguistic quality of the APIs (e.g., Petrillo et al. (2016), Bertolino et al. (2009), Wei et al. (2015) and Rodríguez et al. (2016)) or software documentation (e.g., Treude et al. (2015)). Other works only focused on the structural design of the APIs, e.g., Brabra et al. (2016, 2019).

Although some of the aforementioned approaches dealt with linguistic aspects of REST or cloud computing APIs, in most cases, they only relied on the subjective view of a set of good linguistic practices and recommendations. There is a lack of dedicated approaches that automatically assess the linguistic quality of APIs from IoT providers by detecting both poor and best practices. Other research focused on the analysis of linguistic aspects of the APIs and their documentation, and to the best of our knowledge, our study is the first that focuses on the linguistic design quality of APIs from IoT providers.

Table 8 shows a summary of the comparison between our SARAv2 approach and the related state-of-the-art studies in terms of their goals and methods. In providing a big picture of the comparison: firstly, SARAv2 is a general approach for analysing REST APIs, and the empirical experiment in the current paper is the first study related to IoT APIs. In this aspect, we studied 19 APIs from 18 different IoT providers, where we performed both syntactic and semantic analysis of more than 1100 URIs. We performed the detection of nine linguistic antipatterns and their corresponding nine linguistic patterns to assess the linguistic quality of IoT APIs because we conjecture that poor linguistic quality hinders the consumption, reusability, and maintenance and evolution of APIs. Studies have been performed for Cloud services (*e.g.*, Petrillo et al. (2016, 2018a) and Brabra et al. (2016)) or REST Web services (*e.g.*, Palma et al. (2015), Parrish (2010) and Rodríguez et al. (2016)), which are mostly based on syntactic analysis. However, to the best of our knowledge, SARAv2 is the first study that analyse IoT APIs both syntactically and semantically.

Secondly, our analysis involved 19 APIs from 18 different IoT providers, which is also comparatively higher than any other studies in the literature, *i.e.*, we wanted to investigate a set of APIs from heterogeneous providers to see, on an average, the ratio of well-designed and poorly-designed APIs in terms of linguistic quality.

A final comparison can be made from the perspective of detection accuracy. Our SARAv2 approach performs with an average accuracy of more than 80%. Nevertheless, other studies (as reported in Table 9) show an average precision between 80.9% and 100%. However, these studies are (i) either focus on other types of APIs (*i.e.*, Cloud services or Web APIs) or (ii) the number

13 http://occi-wg.org/about/specification/.

Table 9
Comparison of the detection accuracy with the state-of-the-art approaches.

Study	Tidy/Amorphous URIs	Contextualised/Contextless Resource	Verbless/CRUDy URIs	Hierarchical/Non-hierarchical Nodes	Singularised/Pluralised Nodes	Pertinent/Non-pertinent Documentation	Consistent/Inconsistent Documentation	Unversioned/Versioned URI	Non-standard/Standard URI	Overall Accuracy/Precision
Parrish (2010)	–	–	–	–	–	–	–	–	–	*n/a*
Wei et al. (2015)	–	–	–	–	–	–	–	–	–	91%
Brabra et al. (2016)	–	–	100%	–	–	–	–	–	100%	98%
Petrillo et al. (2016)	–	–	–	–	–	–	–	–	–	*n/a*
Rodríguez et al. (2016)	–	–	–	–	–	–	–	–	–	*n/a*
Haupt et al. (2017)	–	–	–	–	–	–	–	–	–	*n/a*
Palma et al. (2015)	96%	77%	95%	60%	73%	–	–	–	–	81%
Palma et al. (2017)	98%	90%	100%	60%	73%	66%	–	–	–	84%
Petrillo et al. (2018a)	–	71%	–	–	–	77%	–	–	–	74%
Brabra et al. (2019)	100%	–	100%	–	100%	–	–	–	–	100%
This study (SARAv2)	67%	78%	97%	85%	88%	41%	84%	100%	89%	81%

Values that are not reported by the studies denoted by '—'.
The *n/a* refers to the cases where the precision/accuracy are not reported.

of analysed APIs is low (*i.e.*, between 2 and 15 APIs) or (iii) the number of detected linguistic patterns and antipatterns is relatively lower (*i.e.*, between 2 and 28 antipatterns) or (iv) they only perform very fine-grained syntactic analyses (*i.e.*, those for OCCI patterns and antipatterns (Petrillo et al., 2016; Brabra et al., 2016)). Considering the highly semantic nature of our automatic analysis using the SARAv2 approach and the subjective validation of the results using experts, which might significantly differ given the full degree of freedom for deciding patterns and antipatterns, we consider an average accuracy of more than 80% is acceptable in the domain of natural language processing (Shah and Jinwala, 2015).

6. Conclusion and future work

The understandability and reusability are two critical factors for API providers. In the literature, researchers analysed APIs for Web applications and cloud services to assess their linguistic design quality (Petrillo et al., 2016; Arnaoudova et al., 2016; Petrillo et al., 2018a,b; Haupt et al., 2017; Hausenblas, 2011; Brabra et al., 2016). In this study, we assess the linguistic quality of APIs for IoT applications by analysing whether they contain linguistic antipatterns. We proposed the SARAv2 (Semantic Analysis of REST APIs version two) approach and used it to perform syntactic and semantic analyses of REST APIs for IoT applications. The REST-Ling realises the SARAv2 approach as a web application to automate the detection of linguistic patterns and antipatterns.

We utilised the REST-Ling tool to detect nine linguistic patterns and antipatterns. We validated the REST-Ling tool by analysing 1102 URIs from 19 REST APIs for IoT applications and showed its accuracy of over 80%.

From the 19 analysed APIs, we found that all of them organise URIs nodes in a hierarchical manner and only Caret, CubeSensors, and Droplit.io APIs involve syntactical URIs design problems. Moreover, IoT APIs designers, in general, do not use CRUDy terms in URIs, which is a good design practice, but then again, they tend not to use versioning in URI – a poor practice. Also, most designers in IoT use hierarchical organisation of nodes in URIs and document them using consistent language. Further, the ✖*Non-pertinent Documentation* was common in all IoT APIs and the majority of the APIs had ✖*Unversioned URI*. In contrast, most of the APIs followed ✔*Tidy URI* and ✔*Consistent Documentation*.

As we compare the detection of antipatterns across the domains, we found that the APIs for Web applications are highly prone to ✖*Amorphous URI* although carefully implement patterns like ✔*Verbless URI* and ✔*Pertinent Documentation*. We also found that the IoT APIs have very tidy URIs, *i.e.*, follow ✔*Tidy URI* and the nodes in the URIs are organised hierarchically, *i.e.*, follow ✔*Hierarchical Nodes*. On an average, 34% of the URIs from the APIs for Web applications are detected as having linguistic antipatterns, in contrast, 17% of the URIs from the IoT APIs are detected as antipatterns. As for the linguistic patterns, 73% URIs in IoT APIs are well-designed compared to 42% URIs of APIs for Web applications, which suggests that IoT APIs like Amazon AWS, Google Nest, IBM Watson, Microsoft Azure are comparatively more well-designed than the APIs for general-purpose Web applications like Facebook, YouTube, or Instagram.

As future work, we want to apply the SARAv2 approach, thus, the REST-Ling tool, to other IoT APIs. Recently, OpenAPI has been evolved to the industry standard for REST API design and specification. We want to analyse OpenAPI JSON/YAML specifications to assess their design and documentation quality. We also want to investigate two of the patterns and antipatterns further – *Pluralised vs. Singularised Nodes* and *Non-pertinent vs. Pertinent Documentation* – as they are affected more by the cognitive ability of the client developers. We also want to build and include an IoT-specific ontology to perform an improved semantic analysis. Finally, while comparing the detection of SARA and SARAv2, a further extension could be to compare the services from the same company/team (*e.g.*, the REST APIs for Web applications *vs.* IoT APIs for IoT applications from Microsoft or Google) to see whether the difference in the antipatterns is due to the difference of the domain (IoT *vs.* Web) or due to different companies having different API design principles or level of experience.

CRediT authorship contribution statement

Francis Palma: Conceptualization, Methodology, Data curation, Software, Visualization, Writing – original draft. **Tobias Olsson:** Validation, Writing – review & editing. **Anna Wingkvist:**

Writing – review & editing. **Javier Gonzalez-Huerta:** Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We would like to thank Niklas Emevi, a full stack Web developer at Tieto CEM, for taking part in the validation process. We are thankful to Osama Zarraa and Ahmad Sadia for their contributions in developing the tool. We extend gratitude to The Knowledge Foundation, Sweden that partially supported this research through the SHADE HÖG-project 2017/0176. This study was also conducted with the support from Linnaeus University Centre for Data Intensive Sciences and Applications (DISA), Sweden.

References

Abebe, S.L., Haiduc, S., Tonella, P., Marcus, A., 2009. Lexicon bad smells in software. In: 2009 16th Working Conference on Reverse Engineering. Lille, France, pp. 95–99.

Aghajani, E., Nagy, C., Bavota, G., Lanza, M., 2018. A large-scale empirical study on linguistic antipatterns affecting APIs. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 25–35. http://dx.doi.org/10.1109/ICSME.2018.00012.

Arnaoudova, V., Di Penta, M., Antoniol, G., 2016. Linguistic antipatterns: What they are and how developers perceive them. Empir. Softw. Eng. 21 (1), 104–158. http://dx.doi.org/10.1007/s10664-014-9350-8.

Arnaoudova, V., Di Penta, M., Antoniol, G., Gueheneuc, Y.-G., 2013. A new family of software anti-patterns: Linguistic anti-patterns. In: 2013 17th European Conference on Software Maintenance and Reengineering. IEEE, pp. 187–196. http://dx.doi.org/10.1109/CSMR.2013.28.

Arnaoudova, V., Eshkevari, L.M., Penta, M.D., Oliveto, R., Antoniol, G., Gueheneuc, Y.-G., 2014. REPENT: Analyzing the nature of identifier renamings. 40 (5) 502–532. http://dx.doi.org/10.1109/TSE.2014.2312942.

Berners-Lee, T., Fielding, R.T., Masinter, L., 2005. Uniform resource identifier (URI): Generic syntax. In: RFC-Base.

Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M., 2009. Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM, pp. 141–150.

Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent Dirichlet allocation. J. Mach. Learn. Res. 3 (4–5), 993–1022, arXiv:1111.6189v1.

Brabra, H., Mtibaa, A., Petrillo, F., Merle, P., Sliman, L., Moha, N., Gaaloul, W., Gueheneuc, Y.-G., Benatallah, B., Gargouri, F., 2019. On semantic detection of cloud API (anti)patterns. Inf. Softw. Technol. 107, 65–82. http://dx.doi.org/10.1016/j.infsof.2018.10.012.

Brabra, H., Mtibaa, A., Sliman, L., Gaaloul, W., Benatallah, B., Gargouri, F., 2016. Detecting cloud (anti)patterns: OCCI perspective. In: International Conference on Service-Oriented Computing. Banff, Canada, pp. 202–218.

Braun, V., Clarke, V., 2006. Using thematic analysis in psychology. Qual. Res. Psychol. 3, 77–101. http://dx.doi.org/10.1191/1478088706qp063oa.

Budanitsky, A., Hirst, G., 2006. Evaluating WordNet-based measures of lexical semantic relatedness. Comput. Linguist. 32 (1), 13–47.

Fakhoury, S., Arnaoudova, V., Noiseux, C., Khomh, F., Antoniol, G., 2018. Keep it simple: Is deep learning good for linguistic smell detection? In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 602–611. http://dx.doi.org/10.1109/SANER.2018.8330265.

Fellbaum, C., 1998. WordNet: An Electronic Lexical Database. Bradford Books.

Fielding, R.T., 2000. Architectural Styles and the Design of Network-based Software Architectures (Ph.D. thesis). University of California, Irvine.

Fredrich, T., 2012. RESTful Service Best Practices: Recommendations for Creating Web Services. Pearson eCollege, URL http://www.restapitutorial.com/resources.html.

Griffith, T., Steyvers, M., 2006. Probabilistic topic models. In: Latent Se-Mantic AnMysis. A Road to Meaning, Vol. 440. Laurence Erlbaum Associates, Hillsdale.

Haupt, F., Leymann, F., Scherer, A., Vukojevic-Haupt, K., 2017. A framework for the structural analysis of REST APIs. In: 2017 IEEE International Conference on Software Architecture. Gothenburg, Sweden, pp. 55–58.

Haupt, F., Leymann, F., Vukojevic-Haupt, K., 2018. API governance support through the structural analysis of REST APIs. Comput. Sci. - Res. Dev. 33 (3), 291–303. http://dx.doi.org/10.1007/s00450-017-0384-1.

Hausenblas, M., 2011. On entities in the web of data. In: Wilde, E., Pautasso, C. (Eds.), REST from Research to Practice. Springer, pp. 425–440.

Keranen, A., Kovatsch, M., Hartke, K., 2017. RESTful design for internet of things systems. https://tools.ietf.org/html/draft-irtf-t2trg-rest-iot-00, Online; accessed 05 September 2019.

Khamis, N., Witte, R., Rilling, J., 2010. Automatic quality assessment of source code comments: The JavadocMiner. In: Natural Language Processing and Information Systems. Springer, pp. 68–79.

Kolb, P., 2008. Disco: A multilingual database of distributionally similar words. In: Proceedings of 9th Conference on Natural Language Processing. pp. 37–44.

Kolb, P., 2009. Experiments on the difference between semantic similarity and relatedness. In: 17th Nordic Conference of Computational Linguistics.

Levin, G., 2017. REST API, design, guidelines, architecture: 7 rules for REST API URI design. Last visited: October 2019. URL https://blog.restcase.com/7-rules-for-rest-api-uri-design/.

Lu, M., Sun, X., Wang, S., Lo, D., Duan, Y., 2015. Query expansion via WordNet for effective code search. In: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering. Montreal, Canada, pp. 545–549.

Madakam, S., Ramaswamy, R., Tripathi, S., 2015. Internet of things (IoT): A literature review. J. Comput. Commun. 03 (05), 164–173. http://dx.doi.org/10.4236/jcc.2015.35021.

Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S.J., McClosky, D., 2014. The Stanford CoreNLP natural language processing toolkit. In: Association for Computational Linguistics (ACL) System Demonstrations. pp. 55–60, URL http://www.aclweb.org/anthology/P/P14/P14-5010.

Massé, M., 2012. REST API design rulebook. In: Vasa. O'Reilly, p. 114.

Mateos, C., Rodriguez, J.M., Zunino, A., 2015. A tool to improve code-first web services discoverability through text mining techniques. Softw. - Pract. Exp. 45 (7), 925–948. http://dx.doi.org/10.1002/spe.2268.

Microsoft MSDN, 2006. Capitalization Styles, Last visited: October 2019. URL https://msdn.microsoft.com/en-us/library/x2dbyw72(v=vs.71).aspx.

Moha, N., Palma, F., Nayrolles, M., Conseil, B.J., Guéhéneuc, Y.-G., Baudry, B., Jézéquel, J.-M., 2012. Specification and detection of SOA antipatterns. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (Eds.), Service-Oriented Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–16.

OMA, 2012. Open mobile alliance: Guidelines for RESTful network APIs.

Palma, F., Dubois, J., Moha, N., Gueheneuc, Y.-G., 2014. Detection of REST patterns and antipatterns: A heuristics-based approach. In: Service-Oriented Computing. In: Lecture Notes in Computer Science, vol. 8831, pp. 230–244.

Palma, F., Gonzalez-Huerta, J., Founi, M., Moha, N., Tremblay, G., Guéhéneuc, Y.-G., 2017. Semantic analysis of RESTful APIs for the detection of linguistic patterns and antipatterns. Int. J. Coop. Inf. Syst. (IJCIS) 26, 1–37.

Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y.-G., Tremblay, G., 2015. Are RESTful APIs well-designed? Detection of their linguistic (anti)patterns. In: International Conference on Service-Oriented Computing. Goa, India, pp. 171–187.

Panziera, L., Paoli, F.D., 2013. A framework for self-descriptive RESTful services. In: Proceedings of the 22Nd International Conference on World Wide Web. In: WWW '13 Companion, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, pp. 1407–1414.

Parrish, A., 2010. Social network APIs : A revised lexical analysis.

Petrillo, F., Merle, P., Moha, N., Guéhéneuc, Y.-G., 2016. Are REST APIs for cloud computing well-designed? An exploratory study. In: International Conference on Service-Oriented Computing. Banff, Canada, pp. 157–170.

Petrillo, F., Merle, P., Palma, F., Moha, N., Guéhéneuc, Y.-G., 2018a. A lexical and semantical analysis on REST cloud computing APIs. In: International Conference on Cloud Computing and Services Science. Funchal, Portugal, pp. 308–332.

Petrillo, F., Merle, P., Palma, F., Moha, N., Guéhéneuc, Y.-G., 2018b. A lexical and semantical analysis on REST cloud computing APIs. In: Ferguson, D., Muñoz, V.M., Cardoso, J., Helfert, M., Pahl, C. (Eds.), Cloud Computing and Service Science. Springer International Publishing, Cham, pp. 308–332.

Rahman, M.M., Chanchal, R.K., 2015. TextRank based search term identification for software change tasks. In: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering. Montreal, Canada, pp. 540–544.

Riggins, J., 2015. API design considerations for the internet of things. https://www.programmableweb.com/news/api-design-considerations-internet-things/analysis/2015/06/30, Online; accessed 05 September 2019.

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G., 2016. REST APIs: A large-scale analysis of compliance with principles and best practices. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (Eds.), Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings. Springer International Publishing, Cham, pp. 21–39.

Rodriguez, J.M., Crasso, M., Zunino, A., Campo, M., 2010. Improving web service descriptions for effective service discovery. Sci. Comput. Program. 75 (11), 1001–1021. http://dx.doi.org/10.1016/j.scico.2010.01.002.

Shah, U.S., Jinwala, D.C., 2015. Resolving ambiguities in natural language software requirements: A comprehensive survey. SIGSOFT Softw. Eng. Notes 40 (5), 1–7. http://dx.doi.org/10.1145/2815021.2815032, URL http://doi.acm.org/10.1145/2815021.2815032.

Tilkov, S., 2008. REST anti-patterns. Available Online: www.infoq.com/articles/rest-anti-patterns.

Treude, C., Robillard, M.P., Dagenais, B., 2015. Extracting development tasks to navigate software documentation. IEEE Trans. Softw. Eng. 41 (6), 565–581.

Wei, F., Barros, A., Ouyang, C., 2015. Deriving artefact-centric interfaces for overloaded web services. In: International Conference on Advanced Information Systems Engineering. Springer, pp. 501–516.

Wilhelm, R., Seidl, H., Hack, S., 2013. Compiler Design: Syntactic and Semantic Analysis. Springer Science & Business Media.

Francis Palma is an Assistant Professor in the Department of Computer Science and Media Technology at the Linnaeus University in Sweden. He had his Ph.D. from Ecole Polytechnique de Montreal, Canada under the supervision of Dr. Naouel Moha and Dr. Yann-Gaël Guéhéneuc in 2015. His main research interests include Service Oriented Architecture (SOA) Antipatterns and Quality of Service (QoS) for Service-based Systems (SBSs).

Tobias Olsson is a Ph.D. student and working as a Lecturer in the Department of Computer Science and Media Technology at the Linnaeus University in Sweden. His main research interests include Service Architecture and Quality, Analysis of Object-oriented Systems.

Anna Wingkvist is a Research Funding Advisor at the Faculty of Technology, Linnaeus University. Anna had her Ph.D. in Computer Science. She belongs to Grant and Innovation Office, a unit of External Relations.

Javier Gonzalez-Huerta is an Associate Professor in Software Engineering at the Department of Software Engineering at BTH. Javier received his Ph.D. in Computer Science from the Universitat Politècnica de Valencia in 2014. His current research focuses on Asset Management, Technical Debt, and Software Process Improvement, with emphasis on empirical research done in collaboration with industrial partners.