A Mixed-method Approach to Recommend Corrections and Correct REST Antipatterns

Fatima Sabir, Yann-Gaël Guéhéneuc, *Senior Member, IEEE,* Francis Palma, *Member, IEEE,* Naouel Moha, *Senior Member, IEEE,* Ghulam Rasool, and Hassan Akhtar

Abstract—Many companies, e.g., Facebook and YouTube, use the REST architecture and provide REST APIs to their clients. Like any other software systems, REST APIs need maintenance and must evolve to improve and stay relevant. Antipatterns—poor design practices—hinder this maintenance and evolution. Although the literature defines many antipatterns and proposes approaches for their (automatic) detection, their *correction* did not receive much attention. Therefore, **we apply a mixed-method approach to study REST APIs and REST antipatterns with the objectives to recommend corrections or, when possible, actually correct the REST antipatterns.** *Qualitatively, via case studies*, we analyse the evolution of 11 REST APIs, including Facebook, Twitter, and YouTube, over six years. We detect occurrences of eight REST antipatterns in the years 2014, 2017, and 2020 in 17 versions of 11 REST APIs. Thus, we show that (1) REST APIs and antipatterns evolve over time and (2) developers seem to remove antipatterns. *Qualitatively via a discourse analysis*, we analyse developers' forums and report that developers are concerned with the occurrences of REST antipatterns and discuss corrections to these antipatterns. Following these qualitative studies, using an *engineering-research approach*, we propose the following novel and unique contributions: (1) we describe and compare the corrections of eight REST antipatterns from the academic literature and from developers' forums; (2) we devise and describe algorithms to recommend corrections to some of these antipatterns; (3) we present algorithms and a tool to correct some of these antipatterns by intercepting and modifying responses from REST APIs; and, (4) we validate the recommendations and the corrections manually and via a survey answered by 24 REST developers. **Thus, we propose to REST API developers and researchers the first, grounded approach to correct REST antipatterns.**

Index Terms—REST APIs, REST Antipatterns, Recommendations, Corrections, Change History

1 Introduction

Application Programming Interfaces (APIs) are the software, programmatic interfaces that help applications and databases share different functionalities and data [1]. Types of APIs include object-oriented APIs, messaging APIs, and Web APIs. This paper focuses on Web APIs that expose functionalities over the Internet, accessible via different protocols over HTTP, and provided/used by developers and their applications [2].

The number of publicly-available Web APIs has been growing rapidly since 2010 [3], offered through various technologies, including RPC, SOAP, and REST. REST (Representational State Transfer) offers many advantages over SOAP (Simple Object Access Protocol) or RPC (Remote Procedure Call) [3]–[6]. REST is also supported by Web giants, like Facebook, Google, Twitter, or YouTube, to provide access to their APIs [2].

REST Web services—called REST APIs—are Web-based and distributed services that may reside on different servers or even in different organisations. Thus, their evolution is beyond the control of their clients, who may be negatively impacted by changes. They may include poor practices, commonly known as *antipatterns*, as opposed to *design patterns*, which are agreed-upon good practices [7]–[9]. Antipatterns may be introduced during the development activities when developers try to deliver REST APIs under time and cost constraints [10], [11]. They may also appear during the evolution of the REST APIs.

Some studies, e.g., [12]–[15], focused on the automatic detection of REST design patterns and antipatterns. Other studies reported the lack of standardisation for documentation as a source of problems [16], [17]. The violation of REST design is also reported on various online developers' forums by client developers, e.g., regarding status code mismatch, caching problems, or cookies issues.

However, to the best of our knowledge, there exists no approach to help developers in *correcting* their REST APIs. No work focuses on the correction of REST antipatterns in REST APIs after such antipatterns have been identified for two reasons. First, there is a lack of understanding of the evolution of REST APIs and REST antipatterns and, second, the source code of REST APIs is often inaccessible (proprietary). Thus, corrections can only be *recommended*, i.e., direct modification of the actual implementation is impossible.

We apply a mixed-method analysis to understand the evolution of REST APIs and to propose an approach recommending corrections to or correcting REST antipatterns. Figure 1 shows the steps that we follow to build our SOCAR approach. The basis to build our approach are the antipat-

- *Fatima Sabir and Yann-Gaël Guéhéneuc are with the Department of Computer Science, Concordia University, Canada*
  E-mail: fatima.sabir@concordia.ca, yann-gael.gueheneuc@concordia.ca
- *Francis Palma is with the Department of Computer Science and Media Technology, Linnaeus University, Sweden*
  Email: francis.palma@lnu.se
- *Ghulam Rasool is with the Department of Computer Science, COMSATS University Islamabad, Lahore Campus, Pakistan*
  E-mail: grasool@cuilahore.edu.pk,
- *Naouel Moha is with the Department of Computer Science, École de Technologie Supérieure (ÉTS) – Université du Québec, Canada*
  E-mail: moha.naouel@estmtl.ca ,
- *Hassan Akhtar is with the Department of Service Delivery, Ericsson in Islamabad ,Pakistan*
  E-mail: hassan.akhtarr@gmail.com ,

terns and their solutions. We observe that many antipatterns did not have *actionable* solutions and that the community sometimes disagree about existing solutions. Therefore, we first perform an observation study to identify (1) antipatterns that exist and which number evolve in REST APIs and (2) heuristics that are discussed/used by developers to address these antipatterns. We use these heuristics in a second step to build our approach.

First, we perform quantitative and qualitative studies to identify the heuristics. *Quantitatively, via case studies*, we observe the evolutions of major REST APIs and the discussions of REST antipatterns associated with/due to their evolution. We analyse 11 REST APIs, including Facebook, Google, Twitter, and YouTube, over six years by computing code changes and detecting the occurrences of eight REST antipatterns. We thus show that (1) REST APIs and REST antipatterns evolve over time and (2) REST antipatterns disappear thanks to different solutions, which we collect as heuristics. Then, *qualitatively, using discourse analysis*, we analyse developers' forums, report that developers are concerned with the evolution and quality of REST APIs, and collect the heuristics suggested to fix REST antipatterns.

Second, using an *engineering-research approach*, we propose a tooled approach, SOCAR (Service Oriented Correction of Antipatterns in REST), to recommend corrections of/correct the eight previous REST antipatterns. We base our approach on the heuristics collected in the previous quantitative and qualitative studies. Thus SOCAR propose to the REST antipatterns solutions that have been observed, proposed, and–or validated by the community. In addition, we also validate SOCAR recommendations/corrections with practitioners through a survey and report an average precision of 75.90% and recall of 67.72%. We also get a 91.12% agreement ratio for SOCAR based on the results of survey participants.

Thus, the main contributions in this article include:

1) Quantitative studies to understand the evolution of REST APIs antipatterns between 2014 and 2020.
2) Qualitative studies to understand the good practices for REST API design from academia and industry.
3) The SOCAR approach and its tool support[1] for recommending corrections to REST antipatterns.

The rest of the article is organised as follows. Section 2 discusses the state-of-the-art studies on the detection, correction, and recommendation of antipatterns in object-oriented systems and Web services. Section 3 describes our quantitative and qualitative studies of REST APIs and REST antipatterns. Section 4 introduces the SOCAR approach and its steps to investigate the recommendation for correcting antipatterns, while Section 5 reports the experiments and results based on our SOCAR approach. Section 6 presents the validation of our recommendations in the form of a survey while Section 7 discusses our results. Finally, Section 8 concludes with some future directions.

2 Related Work and Background

This section discusses relevant studies on the detection and correction of antipatterns from the literature.

[1] http://www.ptidej.net/downloads/replications/tse20/

2.1 Studies on REST Web Services

REST has gained vast popularity among the Web services community to design REST Web services [17]. Li and Chou [17] noticed that most Web services that claim to use the REST architectural style are not hypermedia driven. They proposed a REST chart model that helps design and describe the APIs without violating basic REST principles, in particular, the hypertext-driven navigation. Another study presented an approach to analysing the APIs based on a machine-readable description [18].

REST APIs description languages like Swagger[2] and RAML[3] received more attention. A recent study regarding the Open API initiative proposed their use as the API description language [19]. Another study reported the description of REST APIs in multiple languages to be transformed into a canonical meta-model. That canonical meta-model acted as a repository to calculate metrics [20]. The authors tested the model for 286 Swagger REST resources and presented the results based on the metrics proposed for the API evaluation [20]. Results showed an average between 9 and 40 REST resources per API but the distribution of resources among the API varies greatly like in Azure, where 61.5% of the resources as read-only. Also, the distribution of REST resources for Google is equally distributed. They also notice that APIs are 'wider than deeper' [20].

The detection of linguistic patterns and antipatterns is reported in literature for REST APIs [13], [21]. The tool called SARA (Semantic Analysis of REST APIs) uses WordNet[4], Stanford CoreNLP[5] along with the Latent Dirichlet Allocation[6] (LDA) topic modeling technique and helps to check the URI nodes to assess their semantic quality [21]. SARA is validated on 11 REST APIs to detect linguistic antipatterns with precision up to 88% compared to DOLAR (Detection of Linguistic Antipatterns for REST APIs) [13] that has an average precision of 79% for the detection of linguistic antipatterns in REST APIs.

Other studies also report the syntactic and semantic design problems in cloud lexicon used by well-known cloud providers like Google Cloud Platform, Open Cloud Computing Interface, and Open Stack [14]. The detection method for Cloud APIs is validated with an approach, CloudLex, for extracting and analysing cloud API lexicon.

All the studies above discuss either best or poor design practices in URIs for REST APIs or detect antipatterns and design patterns in REST APIs. None discusses the evolution of the REST APIs for the correction of antipatterns. Previous work discusses the correction of code smells in object-oriented (OO) systems using techniques from the domains of data mining [22], natural language processing [23], source code metrics [24], and machine learning [25].

2.1.1 Correction of Antipatterns in OO Systems

Software engineers may relate documentation and changes in the source code using a version control system [26]. The percentage of reusable code and examples of such code can be semi-automatically recommended [27].

[2] https://swagger.io
[3] https://raml.org
[4] https://wordnet.princeton.edu/
[5] https://stanfordnlp.github.io/CoreNLP/
[6] https://algorithmia.com/algorithms/nlp/LDA

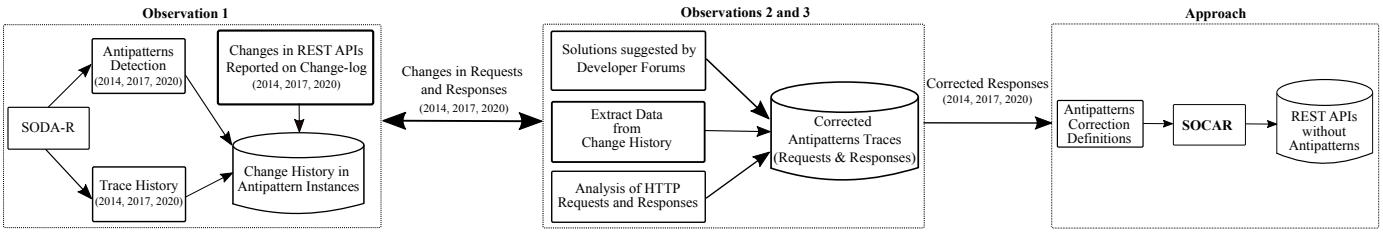


Fig. 1: Observational Study for the Implementation of SOCAR Approach.

Previous work studied the prioritisation of code smells based on recommendation strategies using multiple heuristics [28]. It also report strategies to remove or fix code smells after completing the definition of missing attributes that help to introduce design patterns and maintain code quality [29]. A multi-objective approach was also implemented to improve the coherence with tool support named MORE [29].

The study presented by Terra et al. [30] formalises 32 refactoring recommendations to remove violations related to architectural conformance. It proposes a tool, ArchFix, that recommend refactorings. Researchers proposed refactorings [31], including *Move Method* to another class, with the help of dependencies established by the source method and the target class while keeping a view of static dependencies, with JMove [31].

The ranking strategies and developer perception are also used to improve code quality [32] by removing code smells [33] with the help of refactoring. Recommendation strategies can help inexperienced developers to perform refactoring operations using a monitor running in the background [34].

Another study discusses the use of refactorings to improve system performance vs. those that might negatively impact evolution [35]. The approach is automated with JDeodorant as an Eclipse plug-in to rank suitable refactorings to remove smells. Some researchers use *implicit dependencies* to guide developers in using the most suitable refactorings [36]. Researchers also proposed an intelligent software refactoring bot, RefBot, integrated into version-control systems, like GitHub. Refbot continuously monitors the software repository and can apply recommendations. It was evaluated with the help of a survey [37].

Most of the studies mentioned above are evaluated using industrial case studies or with controlled experiments.

2.1.2 Correction of Antipatterns in Web services

Pautasso [38] proposes solutions to some REST antipatterns in the form of guidelines. Several books discusses avoiding antipatterns in REST APIs [39]–[42]. A recent study studied REST API evolution and its impact on source-code quality [43] through a qualitative study of Facebook, Google Maps, and Twitter. Espinha et al. conducted a case study to investigate the evolution of REST APIs without discussing solutions to the problems faced by API providers during the evolution of APIs [43].

Daigneau [44] addresses the importance of design decisions, focusing on suitable patterns. Most providers allow developers to use old versions for periods of times [44], which impedes tracking problems faced by users.

Different studies report violation of design principles in Web services [45], [46]. Practitioners mostly use the code-first technique to generate source code and then use WSDL generation tools for service interfaces. Mateos et al. [45] propose a tool that assists developers in generating WSDLs with the help Eclipse plug-in called AF-Java2WSDL [45]. They also propose an approach to generate contract-first WSDL documents [46].

A study [47] proposes OO metrics-driven refactorings for code-first WSDL-based services. Early code-first refactoring technique helps avoid antipatterns during the migration phase from traditional systems to SOA-based architecture [48]. Another research proposes using early code refactoring to retrieve the syntactic registries [49] with the help of CBO (Coupling Between Objects), WMC (Weighted Method Complexity), ATC (Abstract Type Count), and EPM (Empty Parameters Methods). They also compare four different Java to WSDL tools and two different registries and shows that refactorings can remove WSDL antipatterns independently of the WSDL generation tools [49].

Rodriguez et al. [50] discuss bad practices that prevent Web services discovery, with 26 professionals. Another study reports WSDL bad practices commonly found in WSDL documents and suggests solutions for these antipatterns [51]. There is an attempt to discover static and dynamic analysis of antipatterns for SOAP services [12] using the SODA-W tool. Another study addresses SOAP antipatterns detection using a parallel evolutionary algorithm [52].

A recent study correlates metrics used to evaluate Web service modularisation using the methodology reported by Kessentini et al. [28] and the approach by Ouni et al. [53]. It is evaluated on a set of 22 Web services provided by Amazon and Yahoo with their publicly-available interfaces [54].

To the best of our knowledge, while studies exist on the detection of antipatterns for Web services, none proposes to correct REST antipatterns in REST APIs to help API clients by improving the comprehensibility, discoverability, usability, and maintainability of REST APIs.

2.2 SOFA Framework and SODA-R

We rely on the SOFA framework (Service Oriented Framework for Antipatterns) to deal with REST antipatterns in REST APIs. The SOFA framework is based on a Service Component Architecture (SCA) and relies on FraSCAti [55] for its run-time support.

An instanciation of the SOFA framework is SODA-R (Service Oriented Detection for Antipatterns in REST). Following IETF guidelines [56], SODA-R implements a list of standard requests, responses, attributes, and status codes.

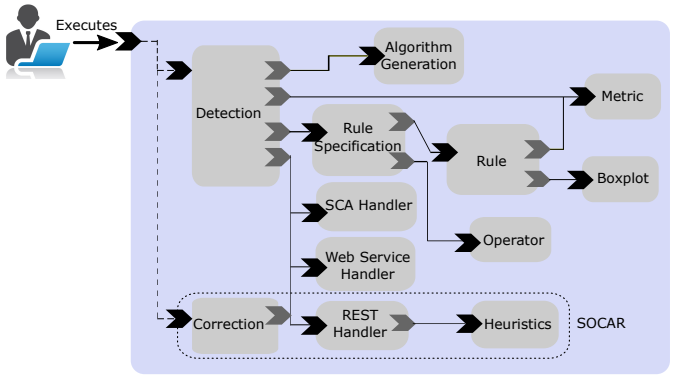


Fig. 2: SOFA Framework.

It allows a requester to call REST APIs and collect the server responses. It also allows analysing these responses, in particular to detect occurrences of some REST antipatterns. It also can modify and present modified responses to the requester. Finally, it can also store HTTP requests and the detection results for REST antipatterns and design patterns instances in CSV (Comma Separated Value) format.

The use of SODA-R requires the URIs of the REST APIs to be called. We carefully craft these URIs following the specifications provided in the documentation of the REST APIs. For example, the URI to obtain a playlist from YouTube is https://www.googleapis.com/youtube/v3/playlistItems?playlistId=1 as per YouTube documentation[7]. In a previous work [12], we used SODA-R to detect 13 REST patterns and antipatterns in documented REST APIs.

2.3 APIs, URIs, Traces, and Trace Histories

For the recommendation of antipatterns correction, we analyse traces of calls to URIs provided by some REST APIs.

REST APIs are build on the HTTP protocol and use HTTP *verbs* (aka *methods*) (typically POST, GET, and DELETE) to access resources identified uniquely by their Uniform Resource Identifiers (URIs). A URI divides into several parts: scheme, authority (userinfo, host, and port), path, and query[8]. The path and query parts form *entity endpoints* (EEPs), which are used with HTTP verbs and URI hosts to create HTTP requests.

Although, technically, REST APIs provide EEPs, in the following, we use the term URIs because we access the EEPs provided by various REST APIs on existing hosts, i.e., www.dropbox.com and, therefore, concretely use URIs to access resources and obtain responses from these accesses.

We call a *trace* the combination of a request and its response for a HTTP access to a URI [57], [58]. A trace includes the request header and body and the response header and body, depending on which HTTP method is being used for the HTTP call. We use SODA-R to obtain traces. By following the IETF guidelines [56] and the documentations of the REST APIs of interest, we ensure that SODA-R collect correct and relevant traces. We put on the companion Web site[1] the list of all the used URIs.

[7]https://developers.google.com/youtube/v3/docs/playlistItems
[8]https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

A trace history is a set of traces for an API collected at different points in time. A trace history is a text file that saves the information of each call performed by SODA-R for each URI. We perform the detection of antipatterns in various years and collection associated trace histories. We thus can compare the trace history of same URIs for different years and different versions.

3 Observational Study

We perform and report the first observational study on REST antipatterns to understand their extent and how developers address them (or not). We select eight REST antipatterns which Palma et al. [13] detected with SODA-R in 11 REST APIs. To the best of our knowledge, this is the first and largest observational study with eight REST antipatterns and 11 REST APIs over the five years.

We present in the next two subsections 3.1 and 3.2 these antipatterns and REST APIs before reporting our observations that (1) REST antipatterns occur and the numbers of their occurrences increase with time (Section 3.3); (2) these occurrences and their increases worry REST developers (Section 3.4); and, (3) these REST developers do correct their APIs to remove these occurrences (Section 3.5).

The results of our observations are heuristics identified either from the actual evolution of antipatterns in REST APIs or from the discussions in developers' community forums. As summarised in Figure 1, we use these heuristics in Section 4 to build our approach.

3.1 Subjects of the Study

We observe the following eight REST antipatterns. Generally, these antipatterns stem from guidelines for REST APIs, which are proposed by different organisations [59], [60] and books [61]. We provide below as many relevant and quality references as possible for each antipattern and provide consensual definitions. Yet, studies reported a lack of industry consistency, e.g., [62], like use of nouns for resource names, which could lead to changes to these definitions as the guidelines evolve and mature.

1. Breaking Self-Descriptiveness (BSD) occurs when developers ignore the standardised attributes, formats, or protocols and use their own customised ones, which breaks the self-descriptiveness or containment of a message header. This poor practice also limits the reusability and adaptability of REST resources [60]. The specification of this antipattern is based on the rules of protocol design [60], [63]. Every resource must have a unique URI, which must provide a list of standardised attributes, media types as a part of API documentation [64].

2. Forgetting Hypermedia (FH) refers to the lack of hypermedia, i.e., not linking resources, which hinders state transitions in REST applications. One indication of this antipattern are missing URL links in the HTTP response, which prevents clients from following the links [61].

3. Ignoring Caching (IC) occurs when REST clients and server-side developers avoid the caching capability. The *caching* is one of the REST constraints. The developers may avoid caching by setting Cache-Control to "no-cache" or "no-store" without an ETag in the response header [61].

4. Ignoring MIME Types (IMT) refers to the practice where server-side developers rely on their own formats, which may limit resource accessibility and re-usability. In general, requested resources should be available in various formats, e.g., XML, JSON, HTML, or PDF [61]. Besides, the response format should be based on the client request [64].

5. Ignoring Status Code (ISC) refers to the practice of avoiding the rich set of pre-defined application-level status codes suitable for various contexts and relying only on common ones, namely 200, 404, and 500, or even use the wrong or no status codes [61], [62].

6. Misusing Cookies (MC) occurs when client developers send keys or tokens in the Set-Cookie or Cookie header field to the server-side sessions. In REST, the session state on the server-side is not allowed and the use of cookies violates this constraint [61], [62].

7. Tunneling Through GET (TTG) occurs when developers rely only on GET for all sorts of actions including creating, deleting, or updating resources. The HTTP GET method is inappropriate for all other actions except retrieving resources [61].

8. Tunneling Through POST (TTP), similar to Tunneling through GET, occurs when the developers depend only on POST for sending all sorts of requests to the server including retrieving, updating, or deleting a resource. In REST, the proper use of POST should be limited in creating server-side resources only [61].

Examples of detected and corrected REST antipatterns are available online[1].

3.2 Objects of the Study

We select 11 common REST APIs in which we detect and observe occurrences of the eight REST antipatterns, which were studied before in the literature [12], [13], [21]. We study 17 different versions of those 11 APIs available online and reported in Table 1.

We choose the URIs to study based on three criteria. First, we choose URIs for which we could collect trace histories using our SOFA framework, as in previous work [12], across the three years of interest: 2014, 2017, and 2020. Second, we choose URIs which participated in some REST antipattern in at least one year of interest found using SODA-R. Third, finally, we choose URIs that we could trace across the years either because (1) their URIs did not change, e.g., https://www.googleapis.com/youtube/v3/playlistItems?playlistId=, or the REST API documentation clearly states how the URIs changed across the years, e.g., from https://api.dropbox.com/1/account/info in v1.0 in 2014 to https://api.dropbox.com/2/users/get_current_account in v2.0 in 2017 and 2020.

The total number of URIs for 11 REST APIs are 308. A statistically significant sample with a 95% confidence level and a confidence interval of 5 would include 172 URIs. We cover all the URIs available from 2014 to 2020 with a sample size 204 to cover the complete population. We conduct a detailed manual analysis of these 204 URIs and the changes associated with these URIs and find 144 URIs that satisfy all of our criteria, as shown in Table 2. We removed Music Graph, Ohloh, Twitter, and Zappos because they do not provide usable change logs, which prevent us to find associated changes.

We performed the data collection in 2014, 2017, and 2020 to observe the changes in the occurrences of antipatterns. We added new versions of the REST APIs in 2017 and 2020 as they came to be released. It was important to collect the traces at different points in time because some REST APIs changed without keeping previous versions, like Alchemy: the "original" v0.9 of 2014 was not accessible anymore in 2017 (even if it was still called v0.9).

3.3 Observation 1: REST Antipatterns Exist and Their Numbers of Occurrences Increase with Time

Table 3 shows the numbers of occurrences of the eight antipatterns detected for the years 2014, 2017, and 2020. It also shows the relative change in antipattern instances for those years in relation to the previous detection year.

We compute the relative change (C_{rel}) in the occurrences of antipatterns, using the following equations, in relation to the actual change (C_{actual}) in the total occurrences of the antipatterns for each REST API from 2014 to 2020, with:

$$C_{actual} = N - N_{ref}$$

where, N is the number of antipatterns in the current detection year and N_{ref} is the number of antipatterns for the previous detection year. Then, C_{rel} is:,

$$C_{rel}(N, N_{ref}) = \left\{ \begin{array}{l} \frac{C_{actual}}{N_{ref}} \times 100\%, \text{for } N_{ref} > 0 \\ 100\%, \text{for } N_{ref} = 0 \end{array} \right\}$$

The relative change cannot be defined for the first detection year 2014 because we performed no detection prior to that year. A *negative* relative change refers to a decrease in antipattern instances. In contrast, a *positive* relative change refers to an increase in antipattern instances compared to the previous detection year.

The relative-change values may vary across the years for a same version, for example for v0.9 of the Alchemy REST API between 2014 and 2017 because some REST APIs did change the behaviours of certain of their URIs without changing their version numbers. Such changes are bad practices in the versioning of REST APIs [65].

We illustrate the detection and evolution of antipatterns with the YouTube API. This API had nine instances of *Ignoring Media Types* (IMT) antipattern in 2014 and 14 instances in 2017, thus, the actual change (C_{actual}) is five and the relative change (C_{rel}) is 56%, i.e., an increase of IMT by 56%. Interestingly, a new detection in 2020 reports ten instances of IMT, i.e., a relative change of -29% that suggests some IMT antipattern instances are removed in 2020.

We now discuss the detection and evolution of antipatterns in Facebook and StackExchange APIs. We choose URIs from Facebook and StackExchange from the year 2014 to 2020 because:

- StackExchange has a complete change log for all the three versions and is available online and the clients can use any one of them. The status code catalog is not complete and only `redirecturi` is labelled as mandatory. The other attributes are optional;
- We observe an increase in 6 URIs for version 2.1 and 8 URIs for version 2.2 compared to version 2.0. Only

TABLE 1: List of 11 REST APIs under Analysis.

REST APIs	Available Versions	Number of Considered URIs	Active Users	URI Changes 2014	2017	2020
Alchemy	v0.9	9	*not available*	-	C	NC
Bitly	v4	15	13,530	-	C	C
Charlie Harvey	v1	12	*not available*	-	C	C
Dropbox	v1, v2	17	500 million	-	C	C
Facebook	v2.7 to v8.0	21	1.94 billion	-	C	C
MusicGraph	v2	19	1 billion	-	C	C
Ohloh	v1.0	7	669,601	-	C	NC
StackExchange	v2.0, v2.1, v2.2	53	345 million	-	C	C
Twitter	v1	25	600 million	-	C	C
YouTube	v3	17	1 billion	-	C	C
Zappos	v1	9	*not available*	-	C	C
11 REST APIs	17 versions	204 URIs				

**Baseline year*
C means that some changes occurred in the query/fields
NC means that no change occurred in the query/fields

TABLE 2: Manual analyses of changes per version of API between 2014 and 2020. We remove Music Graph, Ohloh, Twitter, and Zappos because they do not provide usable change logs.

API Names	Years	Versions	Number of Studied URIs	URIs Added	URIs Removed	Types of Change	Antipatterns Introduced
Alchemy	2014	v0.9	9	-	2	Enhancement	16
	2017	v0.9	9	3	4	Enhancement	31
Bitly	2014	v3	15	-	-	-	7
	2017	v3	15	-	1	Enhancement	24
	2020	v4	15	14	5	New Feature	32
Charlie Harvey	2014	v1	12	-	-	-	8
	2017	v1	12	2	-	Enhancement	12
	2020	v1	12	-	-	-	12
Dropbox	2014	v1	17				38
	2017	v2	17	17	-	Enhancement	45
	2020	v2	17	10		New Feature	23
Facebook	2014	v2.1	21	-	-	-	49
	2017	v2.7	21	7	10	Enhancement	62
	2020	v3.10	21	10	10	Enhancement	67
StackExchange	2014	v2.0	53	-	-	-	73
	2017	v2.1	53	6	2	Enhancement	78
	2020	v2.2	53	5	1	Enhancement	80
YouTube	2014	v3	17	-	-	-	21
	2017	v3	17	26	-	Enhancement/New Feature	37
	2020	v3	17	58	-	Enhancement/New Feature	22
	3 years	13 versions	144 URIs				

a few changes are observed for the TTG and TTP antipattern for version 2.1 and 2.2;

- Facebook has seven versions running simultaneously for two years, i.e., currently, they are using version 9.0 although version 3.10 is still operational and Facebook is still allowing clients to migrate to the newer versions;
- We selected only those URIs that are available in versions from 2.10 to 3.10 such that they are continuously evolving;
- It was not possible to perform versioned analysis of request/response because Facebook provides the calls to resources using non-versioned URI;
- Traces of Facebook version number is neither found in CSV files nor trace logs;
- We could identify common resource URIs available in both years 2014 and 2017 from the APIs online change log, as listed in Table 1. The antipattern evolution along the version history of Facebook and StackExchange is available in Table 3.

We analysed the v2.1 in 2014, v2.7 in 2017, and v3.10 in 2020 for Facebook API. For StackExchange, we analysed v2.0, v2.1, and v2.2 for the three years in 2014, 2017, and 2020. We randomly selected 21 URIs for Facebook and 53 for StackExchange to collect their trace information in which we identified the presence of antipatterns.

Actual and relative changes for the version history of Facebook and StackExchange are available in Table 3. The columns BSD, FH, IMT, ISC, IC, MC, TTG/TTP shows the occurrences and changes in the occurrences of antipatterns for each year. For example, the detection results for *Forgetting Hypermedia* (FH) antipattern in the trace history of YouTube show us the changes in the antipatterns instances across different versions for the years 2014, 2017, 2020. This

TABLE 3: Actual and Relative Changes in Antipatterns between the year 2014 and 2020.

API Names	Year	Version	BSD	C_{rel}	FH	C_{rel}	IMT	C_{rel}	ISC	C_{rel}	IC	C_{rel}	MC	C_{rel}	TTG/TTP	C_{rel}
	2014	0.9	0	n/a	1	n/a	2	n/a	1	n/a	7	n/a	0	n/a	5	n/a
Alchemy	2017	0.9	9	100%	1	0%	2	0%	1	0%	9	29%	0	0%	9	80%
	2020	0.9	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	2014	3	0	n/a	2	n/a	3	n/a	0	n/a	0	n/a	0	n/a	2	n/a
Bitly	2017	3	0	0%	5	150%	15	400%	0	0%	0	0%	0	0%	4	100%
	2020	4	14	100%	0	-100%	14	-7%	0	0%	0	0%	0	0%	4	0%
	2014	1	4	n/a	0	n/a	4	n/a	0	n/a	0	n/a	0	n/a	0	n/a
Charlie Harvey	2017	1	12	200%	0	0%	0	-100%	0	0%	0	0%	0	0%	0	0%
	2020	1	12	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
	2014	1	12	n/a	9	n/a	0	n/a	0	n/a	12	n/a	0	n/a	5	n/a
Dropbox	2017	2	17	42%	14	56%	0	0%	0	0%	8	-33%	0	0%	6	20%
	2020	2	17	0%	0	-100%	0	0%	0	0%	0	-100%	0	0%	6	0%
	2014	2.1	21	n/a	19	n/a	8	n/a	0	n/a	0	n/a	0	n/a	1	n/a
Facebook	2017	2.7	21	0%	21	11%	12	50%	0	n/a	4	100%	4	100%	0	-100%
	2020	3.10	21	0%	21	0%	21	75%	0	n/a	4	0%	0	-100%	0	0%
	2014	2	0	n/a	1	n/a	2	n/a	1	n/a	7	n/a	0	n/a	5	n/a
Music Graph	2017	2	9	100%	1	0%	2	0%	1	0%	9	29%	0	0%	9	80%
	2020	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	2014	1	3	n/a	0	n/a	0	n/a	0	n/a	1	n/a	3	n/a	0	n/a
Ohloh	2017	1	0	-100%	0	0%	7	100%	0	0%	0	-100%	0	-100%	0	0%
	2020	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	2014	2.0	0	n/a	19	n/a	53	n/a	0	n/a	0	n/a	0	n/a	1	n/a
	2017	2.0	0	0%	19	0%	53	0%	0	0%	0	0%	0	0%	1	0%
	2020	2.0	0	0%	19	0%	53	0%	0	0%	0	0%	0	0%	1	0%
	2014	2.1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
StackExchange	2017	2.1	0	n/a	24	n/a	0	n/a	0	n/a	0	n/a	0	n/a	0	n/a
	2020	2.1	0	0%	24	0%	53	100%	0	0%	0	0%	0	0%	1	100%
	2014	2.2	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	2017	2.2	0	n/a	26	n/a	53	n/a	0	n/a	0	n/a	0	n/a	1	n/a
	2020	2.2	0	0%	26	0%	53	0%	0	0%	0	0%	0	0%	1	0%
	2014	1	10	n/a	3	n/a	9	n/a	0	n/a	0	n/a	0	n/a	0	n/a
Twitter	2017	1	25	150%	6	100%	25	178%	6	100%	14	100%	0	0%	2	100%
	2020	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	2014	3	9	n/a	3	n/a	9	n/a	0	n/a	0	n/a	0	n/a	0	n/a
YouTube	2017	3	17	89%	3	0%	14	56%	0	0%	3	100%	0	0%	0	0%
	2020	3	12	-29%	0	-100%	10	-29%	0	0%	0	-100%	0	0%	0	0%
	2014	1	7	n/a	0	n/a	0	n/a	0	n/a	0	n/a	0	n/a	0	n/a
Zappos	2017	1	9	29%	0	0%	0	0%	0	0%	0	0%	0	0%	1	100%
	2020	1	0	-100%	0	0%	9	100%	0	0%	0	0%	0	0%	1	0%

antipattern is due to service responses that do not contain any link or the absence of link for *location* attribute. They reduce API usability by preventing clients to follow links.

While we study the REST APIs that evolved, for example, for StackExchange from version 2.0 to 2.2, we found significant change in the *Forgetting Hypermedia* antipattern, i.e., from 19 in v2.0 in the year 2014 and 2017 to 26 in v2.2 in the year 2020. For example, for the HTTP call /2.0/me/write-permissions has no instance of this antipattern in API version 2.0 but the next version 2.1 has one instance. The instance is also not removed in version 2.2.We also noticed an increase in the *Forgetting Hypermedia* antipattern from version 2.0 to 2.2. This information could also be confirmed from the trace history of StackExchange API available on our companion Web site[1]. Many URIs do not change across versions, i.e., do not introduce any new antipatterns.

Summary on Observation 1: We found that REST antipatterns evolve, for example because of service migration to different hosting services. Trace history collected using SODA-R provides antipatterns evolution along version history, as shown in Figure 3. We observed relative changes for antipatterns in APIs from the year 2014 to 2020. APIs documentation or online change logs are not given by some API providers or have little information for client developers to use APIs correctly (e.g., Bitly, Ohloh, Music Graph, Charlie Harvey, Zappos).

3.4 Observation 2: REST Antipatterns Do Worry API Developers

Previous work used public, professional developers' forums to study developers' problems and solutions [66], [67]. StackOverflow is a widely used forum for discussing development issues. It provides tags on questions and reputation based on positive and negative votes [68].

A StackOverflow data dump is publicly available under the Creative Commons License, which we downloaded for the year 2020 and the topic "software engineering", which include 2,284,42 posts. (The data dump provides five XML documents: Posts, Votes, Comments, Users, and Badges. We are interested by the Posts to identify problems related to REST APIs and their solutions by the community.)

The Posts data contains questions with unique IDs, creation dates, answer counts, comment counts, and associated tags. We follow the following steps to identify posts discussing REST API antipatterns and any proposed solution:

1) We filter the posts using *related tags*, i.e., *rest* and *restful-architecture* [69], yielding 87 and 157 respectively.
2) We consider only recent posts of no more than five years and with up-votes for their answers greater than ten, yielding 52 posts.
3) The questions and answers of the 52 posts are manually evaluated by the authors based on their titles, reported problems, and answers, yielding 31 posts.
4) We apply a two-step verification process to manually verify each post question and answer to select the best solution related to REST API design.

As a result, we find 31 relevant posts that discuss problems with REST API design and their solutions. The data of all posts and the selected posts are available on our companion Web site[1].

Table 4 reports different problems faced by developers when service providers do not provide enough documentation. It also summarises the answers proposed on the developer forum to each of these problems.

In the posts, developers highlight the importance of best practices in REST API design. Some best practices consider guidelines proposed by academia and available in the literature [8], [9]. The definitions of REST antipatterns also follow the guidelines proposed on professional forums [70]–[72] and in the literature [8], [9].

While documentation and change logs may help client applications use REST APIs, researchers reported the lack of quality documentation as a major hurdle to use APIs [16], as illustrated by a case study conducted at IBM [16]. We also observe the same problem while creating our requests to the REST APIs studied here, i.e., the REST API documentation does not provide complete guideline for status code and MIME type (e.g., Charlie Harvey and Ohloh). Complete information of status code, caching time limit, and required attributes, standardised header list is also often not available. However, Facebook, Dropbox, Twitter, and YouTube provide complete information with examples.

Summary on Observation 2: Developers are concerned with REST antipatterns. On average, there are 107K views for posts associated with REST antipatterns and the use of good design practices in REST APIs. Some of the posts also provide complete guidelines for the corrections of REST antipatterns, i.e., Status Code good practises. There is a need to support clients and providers to improve REST APIs with solutions to REST antipatterns. Most of the posts refer to the answers based on guideline provided by Internet Engineering Task Force (IETF) [60], [73], [70], [71]. REST APIs are also used by commercial web applications that use cookies but IETF set rules based on time and also recommend to provide complete series of Status code.

3.5 Observation 3: REST Antipatterns Are Corrected by Their API Developers

We investigated the heuristics applied by developers to correct antipatterns for REST APIs.

Facebook, YouTube, and Dropbox provide a complete list of information of each version change along with the migration guide from one version to another. Table 2 shows the type of changes associated with the year 2017 and 2020, compared to 2014, as Enhancement or New Feature. Such analyses of the change logs also helps us to form the correction heuristics of REST antipatterns.

We illustrate developers' corrections with examples from Alchemy, Bitly, Dropbox, Facebook, and YouTube APIs.

3.5.1 Alchemy API

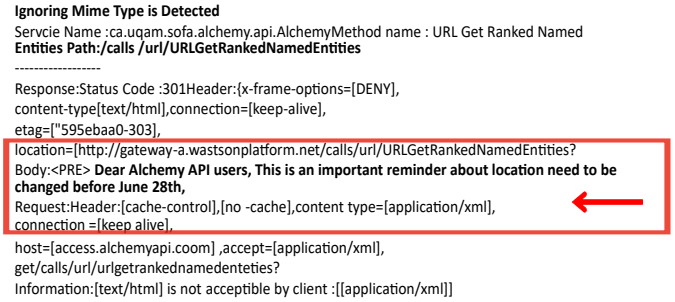


Fig. 3: Migration Traces of Alchemy API.

Alchemy is the only API that has a migration history for resources and servers on the IBM Watson platform. We invoke seven URIs from Alchemy API and collect their responses in 2014 and 2017 to observe the changes in response. The Alchemy API was not available in 2020 anymore.

We called 9 URIs of the Alchemy API in 2014 and 2017. The selected URIs remained the same across the years but showed different responses (and antipatterns). Alchemy had 16 URIs in June 2014. Two URIs were removed in late 2014, replaced by 3 more new URIs. We observed 17 URIs in June 2017. However, four URIs were removed in late 2017 and Alchemy had 13 URIs in late 2017.

The number of instances of the BSD, IC , TTG, and TTP antipatterns increased due to changes in the location attributes, cache-control information, and the addition of two new POST and one new GET method, not following

TABLE 4: Issues Reported on professional Developer Forum.

Antipattern	Question	Answer
BSD	Contract and protocol to communicate?	Link should be provided
	Use of Default value	Follow RFC standard [70]
FH	Link relation value	Link based on client metadata
	Hypermedia rel value setting	Apply rel value for back end changes
	Status code in HATEOS	Follow RFC standard [70]
IMT	REST semantic link vs. URI	Link should be provided
	Best format to use REST design	Use XML and JSON as per content request
	Preferred internet media type	Support (JSON/XML)
	REST biggest adoption blocker?	Support JSON
	Tradeoff between request vs. response	Prefer JSON,Decide content header for request
	Formate to get response	Client can specify formate
IC	Session Violates REST?	Yes
	Explicitly set Cache state	Cache response along response code
	How long I can cache?	Set cache explicitly
	Cache Http response	Set time
	Ignore HttpCache header?	Set time or state as no-cache
	How to use Cache?	Cache control: no Cache,no store
ISC	Status code in HATEOS?	Follow RFC standard [70],response code 409
	Response in Status code	400 for bad request
	Serve client request	Http 4xx
	Response in Status code	Response along status code description
	Response HTTP message	Follow RFC standard [71]
	HTTP status code for request limit	Follow RFC standard [71]
MC	Authorization header?	Avoid Cookies
	Authorization header?	Apply based on time
TTP	What exactly are link relation value ?	Not use POST for CRUD operations
	Represent action verb	Use Noun
	REST endpoint design	Follow RFC standard [70]
TTG	When to use POST?	Follow RFC standard [70]
	Real time access to server?	'Get' to retrieve resource
	POST response format?	Specify response as per Operation type

good practices, as reported in Table 6. The Alchemy API, after its migration in 2020, has more non-standard attributes in its headers than in 2014.

We detail the URI /calls/url/URLGetText and changes in the evolution history for the *Ignoring MIME Types* antipattern [59]. Figure 3 shows the change history of the traces collected using SODA-R [12] for the detection of *Ignoring MIME Types* antipattern in /calls/url/URLGetText. The complete trace-log is available online[1].

We consider Alchemy API as an example due to the migration phase from HTTP to HTTPS by changing its host from www.access.alchemyapi.com to www.gateway-a.watsonplatform.net. We expected some improvements and changes in the APIs and, indeed, the *Ignoring MIME Types* antipattern disappeared during the migration. We observed that the evolution of the domain name did not affect the instances of *Ignoring Status Code* and *Misusing Cookies* for /URLGetRankedNamedEntities. We studied the online documentation and it shows no change in attributes or methods for /URLGetRankedNamedEntities while /TextGetRankedNamedEntities has new attributes, using access-control-allow-origin, content-security-policy, x-content-type-options, x-alchemyapi-status, and strict-transport-security.

3.5.2 Bitly API

Bitly is currently running its version 4.0, which relies on OAuth 2.0 with an SSL implementation (Secure Sockets Layer). Only limited details are available in the documentation regarding the API implementation. A well-documented API should describe all the underlying resources, status codes, and HTTP methods. Bitly only describes the status codes 200, 400, 403, 500, and 503. We implemented all the status codes [56] and made HTTP requests to collect responses from Bitly, as we did with the other REST APIs.

Table 3 shows the evolution of the REST antipatterns from 2014 until 2020. We did not observe any changes for *Ignoring Caching*, *Misusing Cookies*, and *Ignoring Status Code*. The *Ignoring MIME Type* antipattern has a slight change in 2020 with the release of a new version of the API, i.e., version 3. The *Tunneling* antipattern has four instances. We found a small change in the instances of the *Forgetting Hypermedia* antipattern, removed in 2020. This change might be due to the absence of the *location* attribute in the response meta-data of Bitly API. The resource URIs most impacted by this antipattern are /link/click, /shorten, /bitly_pro_domain, /user/tracking_domain, and click/user/tracking_domain_list. This antipattern then evolved in July and impacted one more resource URI, /link/encoders_by_count. Bitly API resources changed in 2020, i.e., the GET method is now used to retrieve resources in multiple forms. Previously, in 2017, Bitly developers relied more on the POST method. The use of both GET and POST methods leads to the removal of the *Tunneling* antipattern.

3.5.3 Dropbox API

The changes that Dropbox API applied from v1 to v2 illustrates the correction of the *Ignoring MIME Types* antipattern, which is introduced when REST developers do not provide multiple representations for a resource, forcing clients to accept a single format, like JSON or XML. Dropbox, like Charlie Harvey and Ohloh, supports JSON as default MIME

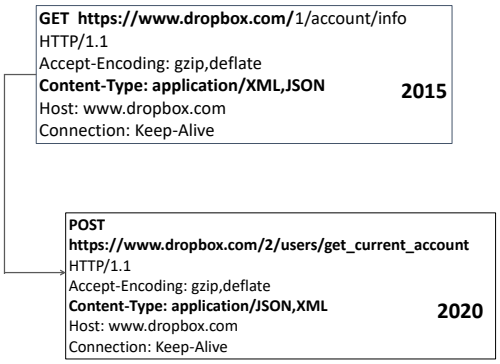


Fig. 4: Migration Traces of Dropbox API.

type but other MIME types, like XML and YAML, are also supported if requested by clients.

We studied the trace history of the Dropbox REST API for URI /1/account/info associated to version 1, which changed into /2/users/get_current_account in version 2. Dropbox offered JSON and XML as MIME types in 2014. They still provide multiple resource representations in 2020. Thus, there is no instance of the *Ignoring MIME Types* antipattern in 2017 and 2020. Figure 4 shows the complete information of traces from version 1 of 2014 to version 2 available in 2020.

3.5.4 Facebook API

We observe changes with each version of Facebook APIs from 2014 to 2020. Facebook supported non-versioned calls until v5.0, which is not recommended by professional developer forums. It supports versioned calls after version 6.0. There are no instances of the *Ignoring Status Code* antipattern in any version. We observe an increase for *Ignoring Caching* for the years 2017 and 2020 due to changes in Facebook marketing policy, now supporting *cache-control as private*.

3.5.5 YouTube API

YouTube has a long change history and continuously evolved during the past few years. We observe changes in the instances of the *Breaking Self-descriptiveness*, *Forgetting Hypermedia*, *Ignoring MIME Types*, *Ignoring Caching*, and *Tunneling* antipatterns between May 2014 and September 2020. We found the *Ignoring Cache* antipattern across many resource URIs over that period. The reason for the evolution of *Ignoring MIME Types*, *Ignoring Caching*, and *Tunneling* is that /video/rate was not available in June 2017. However, the response status did not provide information of the change in URI, which was also not available in the client-side credential details. Some of its attributes, e.g., recordingDetails.locationDescription, recordingDetails.location.latitude, and recodingdetail.location.longitude, were deprecated in 2017.

There is also a change in /video/rate resource in 2020 due to new policy. Videos uploaded after the 28 July 2020 are restricted while they undergo an audit. These changes impacted the response of /video/rate. We also found non-standard attributes, e.g., alt-svc from /activities, /playtimes, and /guidecategories, which cause the introduction of the *Breaking Self-descriptiveness* antipattern.

Table 3 shows the evolution of the REST antipatterns between 2014 and 2020 for YouTube. Instances of the *Ignoring MIME Types* antipattern also exist in /activities, /playtimes, and /guidecategories, for which these URIs *only* return JSON while they should also return XML and other formats. The instances of *Ignoring MIME Types* decreased in 2020. Yet, YouTube still relies on a single representation but we can parse and convert it into text and XML, thus removing these URIs from the list of *Ignoring MIME Types* antipattern.

Summary on Observation 3: While we observe and report that REST APIs change in time and that the numbers of antipatterns and their occurrences also change, we could not find direct evidence in the change logs that developers targeted these antipatterns. However, REST APIs evolve and change with new practices and technologies, which changes the number of antipatterns. For example, multiple MIME types are still supported by some API providers, i.e., Dropbox, but the majority of API providers offer a single MIME type. As other example, caching changed with different version of the Facebook APIs, from version 2.7 to 3.1, against guidelines. Thus, it seems that developers do correct REST antipatterns, albeit not explicitly.

4 SOCAR Approach

We develop SOCAR (Service Oriented Correction of Antipatterns in REST) as a recommendation/correction system that identifies occurrences of REST antipatterns and recommend corrections for/correct these occurrences.

In a preliminary step, we encode the corrections observe in the previous observation study as recommendations of corrections or actual corrections (when possible). Then, SOCAR divides into two steps: first, it applies SODA-R to some REST API to detect occurrences of some REST antipatterns and, second, it proposes corrections or correct these occurrences.

4.1 Preliminary Step: Definition of the Corrections

Input: Qualitative study (in Section 3).
Output: Correction recommendations/Corrections.
Description: The trace history of the APIs can be used to decide the possible corrections for REST antipatterns. The properties used to correct REST antipatterns are also available in the literature [12], [59]. The correction heuristics are based on the changes observed for each URI based on the observational study, which informs our correction heuristics. Some of the corrections are recommendations, e.g., when responses are not parsed as per clients' choice, like *Ignoring Cache*, while others are corrections without changing the response, e.g., *Ignoring Status Code*.

For example, for recommending the correction of *Breaking Self-descriptiveness* antipattern, for each API we identify the list of non-standard header descriptions [56] and use a *remove()* correction in case of the detection of non-standard header attributes in the HTTP response from the server to display the corrected response headers to the clients.

In another example, for recommending the correction of *Ignoring Status Code* antipattern, we implement the status code list in the correction definition of *Ignoring Status Code*

antipattern and applied the *add()* correction on the response header field that helps clients to identify the correct status for HTTP calls. It is also possible to have the wrong status description, which, if found, is corrected after applying the *replace()* method. We do this by removing the improper 'error description' with the proper description. All the corrections are implemented in SOCAR for the correction of REST antipatterns.

We develop correction definitions after a detailed analysis of the attributes in HTTP requests and responses collected from the trace history of REST APIs.

We implement different operations in SOCAR for the corrections of REST antipatterns in REST APIs, as shown in Table 5. We add missing attributes using evolution history in the HTTP request and response to correct antipatterns. Table 6 reports the correction rules applied for each antipattern related to their associated properties.

TABLE 5: SOCAR Corrections/Recommendations

Operations Name	Changes Required	Descriptions
Fixing error code	Response	API's response status change
Enhancement	Request, Response, Body	API changes its version or domain
New feature	Request, Response	API has an additional functionality
Correction	Response	Correction of Response through changes to attributes

Based on the recommended corrections in Table 6, we proposed correction heuristics. Algorithms 1 and 2 show the correction heuristics for *Ignoring MIME Types* and *Forgetting Hypermedia* antipatterns, respectively; the most common REST antipatterns in REST APIs [12], [73].

Input: *request-metadata, response-metadata*
Output: *Content Type Design Pattern*
request-metadata ⟵ *xml, json, pdf, html, ...;*
response-metadata ⟵ *xml, json, pdf, html, ...;*
if *request-metadata.containskey()= "accept"* **then**
  if *request-metadata* ≠ *response-metadata* **then**
    response-metadata.add (request-metadata);
    "Ignoring MIME Types Corrected based on Client Request"
  end
  "Ignoring MIME Types Instances Removed"
end
"Content Negotiation Pattern Detected"

Algorithm 1: Correction of *Ignoring MIME Types* REST Antipattern.

In Algorithm 1 for *Ignoring MIME Types* antipattern, the heuristic is checking whether the *accept* field is present in the request header. In the case of presence, if the response header does not match the request header, i.e., the responded MIME type is not the same as the MIME type request by the client, the response header is recommended to be updated with an appropriate MIME type. The appropriate MIME type here one that is requested by the client in the request header. The correction for *Ignoring MIME Types* antipattern introduces the *Content Negotiation* REST pattern, which is a good design practice in REST.

Input: *HTTP Method, Format, Location*
Output: *Entity Link Design Pattern.*
HTTP Method ⟵ *GET, PUT, POST, DELETE;*
Format ⟵ *XML, JSON, PDF, RDF, HTML, ...;*
URI ⟵ *HTTP Method + Location + Format;*
response ⟵ Access *URI;*
if *response.getstatus()="4K" or response.getstatus()="5K"* **then**
  Remove from detection and correction;
end
if *response.getBody() and response.getmetadata()* ≠ *Null* **then**
  if *checkLinksBody()* ⟵ *XML, JSON, PDF, RDF* **then**
    if *checkLinkMetaData()* ⟵ *URI* **then**
      Link detected ; **if** *checkLinkMetaData().contains* ≠ *URI* **then**
        linkmetadata.add(URI);
        "Addition of link in Response meta data"
        "Forgetting Hypermedia Antipattern Correction"
      end
    end
  end
end
"Entity Linking Pattern detected"

Algorithm 2: Correction of *Forgetting Hypermedia* REST Antipattern.

We cannot change the MIME-Type on the server because we do not have access to source code and data. Also, professional developers are divided on the use of MIME type, either JSON or XML. Therefore, we build a parser that first identifies and then parses the response and then converts it into the appropriate MIME type, following a client's request, either in JSON or XML. Table 6 also provides options for the correction of the IMT antipattern based on a client's choice.

The heuristic in Algorithm 2 for *Forgetting Hypermedia* antipattern checks for the absence of resource links in the response body or response header. In the absence of such links, e.g., the response header does not have a location field, the heuristic recommends to add a resource URL to the response header. This correction introduces the *Entity Linking* REST pattern, which is another good design practice.

4.2 Step 2: SODA-R to Detect REST Antipatterns

Input: REST APIs for their dynamic invocation via client requests.
Output: Detected instances of REST design patterns and antipatterns from the client- and service meta-data.
Description: The SODA-R approach [12], proposed by Palma et al., checks the instances of antipatterns in REST APIs. SODA-R implements the detection heuristics of eight REST antipatterns and performed their detection from popular REST APIs like Facebook, YouTube, and Dropbox. There is a need to check the antipattern instances before implementing their correction heuristics. This step also helps us study the evolution of REST antipatterns, i.e., if

TABLE 6: Antipatterns Correction Rules.

Anti-patterns	Properties	Recommendations From Literature	Recommendations From Practice	Consensus Applied	Effect
IMT	Accept, Content Type	json,xml [64]	Prefer json	Content selection as per user choice	*Content Negotiation* Pattern
IC	Cache-Control, ETag	Add Cache Control, Generate Unique E-Tag for Request [61]	Cache control='no cache'	Content selection as per user choice	*Response Caching* Pattern
FH	http-methods, entity link, location	add links, metadata info, status code [61]	dynamic link as per developer choice	Content selection as per user choice	*Entity Link* Pattern
BSD	request and response header field	Remove non-standard headers from Response [60]	follow IETF guideline [56]	Consensus matched	Antipattern removed
ISC	http method, status, standard status code description,	Status code number and description change, replace method for code description and number [61] [62]	Follow complete description of Status Code [56]	Consensus matched	Antipattern removed
MC	Cookie, Set Cookie	Remove Set-Cookie, cookie from response metadata [61] [62]	Set-Cookie as per company policy	Content selection as per user choice	Antipattern removed
TTP	http-method, request-URI	remove access, update and delete from resource URI [61]	Follow guidelines using POST [72] [59]	Consensus matched	Antipattern removed
TTG	http-method, request-URI	remove access, update and delete from resource URI [61]	Follow the guidelines using GET [72]	Consensus matched	Antipattern removed

antipatterns have increased or decreased from the recent studies. Furthermore, this may help us to check changes in instances of antipatterns associated with each REST API.

SODA-R provides detection results for design patterns and antipatterns and the trace history to check the attributes in HTTP requests and responses for REST APIs. The correction mechanism requires proof that services are improved and that the number of antipatterns instances is either decreased or equal to zero. A study showed the relationship between code smells and design patterns: if the services are improved, then the total number of design patterns for a specific Web service increases after the correction of antipatterns [74].

4.3 Step 3: SOCAR to Provide/Apply Heuristics

Input: Correction recommendations.
Output: The recommended REST design pattern instances and the related REST antipatterns that are removed.
Description: REST APIs evolve rapidly, which is evident from their online change history available in Table 1. The algorithms of correction heuristics are applied based on the data collected from trace history along with the recommendation from developer forums. Table 6 reports the results along with properties collected from trace history and corrections suggested by academia along with the consensus of the developer forum. In Table 6, the *Recommendation from Literature* column provides information regarding the definition of antipatterns proposed by academia. The column *Recommendations from Practice* shows the developer views regarding the correction definition for antipatterns removal. The column *Consensus Applied* shows the SOCAR implementation for each REST antipattern correction approach. Corrections are also performed for client in case changes are required in the *Response*. The SOCAR approach uses the recommendation of academia and professional developer and in case the consensus is not matched, then the client developer has the right to apply the definition of correction heuristics proposed by the industry or academia. We name this selection as *Content Selection as per User Choice*.

SOCAR recommends corrections for Ignoring Mime Type and Ignoring Status Code by parsing the response, modifying it, and sending the modified on to the client. We developed a parser to parse the MIME type and provide input text file to map the status code of response and replace with correct status code as per IETF guidelines. We provide correction recommendations to Breaking Self-Descriptiveness, Misusing Cookies, Cache Control, and Tunneling because they cannot be simply fixed for the client. (Corrections/Recommendation of corrections are available on-line in the companion Web site[1].

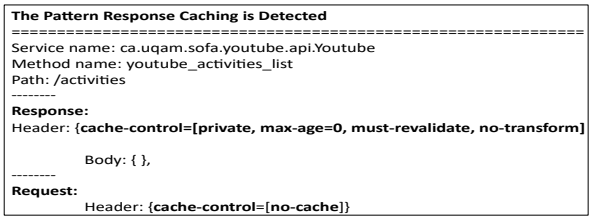


Fig. 5: Traces of Design Pattern *Response Caching*.

Figure 5 shows a trace collected from YouTube and that shows a Response Caching design pattern, with max-age and re-validate attributes. However, developers from the industry and academia have different opinions on caching and also suggest using No-Cache, thus, the SOCAR approach allows the client developers to choose the correction heuristic for *Ignoring Caching* antipattern between setting "cache-control" as "no-cache" or setting "cache-control" as "public" or "private" and "max-age" as zero.

5 Experiments

We performed a series of experiments using our recommended corrections in Table 6. As objects, we consider the 11 REST APIs in Table 1. The eight REST antipatterns that we consider as subjects are listed in Section 3.1. Through the experiments, we want to show the accuracy of the SOCAR approach in terms of precision of the corrections, i.e., how many of the detected antipattern instances could be corrected.

5.1 Process

We analysed the complete trace history of each REST API from 2014 to 2020. We investigated the changes in the occurrences of each antipattern. Each API shows the introduction or removal of antipatterns based on different attributes collected from trace history.

For the experiments, first, we invoke the client API requests for the REST APIs based on the required information for calling a specific function, e.g., using profile API of Facebook or rating video using YouTube API. As a client, we receive requests with multiple attributes representing the resource information from API providers. The requests are analysed based on the definition of design REST antipatterns. SOCAR checks the requests invoked by the client and responses received from the API providers. We then analyse the requests and responses based on the corrections for each antipattern in Table 6. The corrections also consider the recommendation collected from the observations. The corrections are recommended based on the definition reported in the literature [12], [13], and collected from developers' forums as in Section 4. After applying the recommended correction, SOCAR provides the instances of the corrected antipatterns.

In the following, we discuss the recommended corrections for two antipatterns that we apply during our experiments, namely for *Ignoring MIME Types* and *Forgetting Hypermedia*. We choose these two antipatterns because they are the most common REST antipatterns in REST APIs [12], [73].

5.2 Corrections for Ignoring MIME Types Antipattern

In Algorithm 1, we apply the heuristic for the correction of *Ignoring MIME Types* REST antipattern. Figure 6 shows the evolution of *Ignoring MIME Types* antipattern along with recommendation of corrections applied by SOCAR for Alchemy API. We also obtain similar traces for Facebook regarding their policy of migration from one version to another. However, no such information is found for Dropbox in its HTTP response body. Instead, they show such information on their developer page. Also, no such information for YouTube was available as they are running version 3 for the last four years with a large set of change history, as reported in Table 1.

We remove the *Ignoring MIME Types* REST antipattern by modifying the response as per the client's request. In Algorithm 1, to remove the *Ignoring MIME Types* antipattern, we examine the client's request and match the response after adding the resource representation dynamically to the response, as per the client's request. For example, if a client

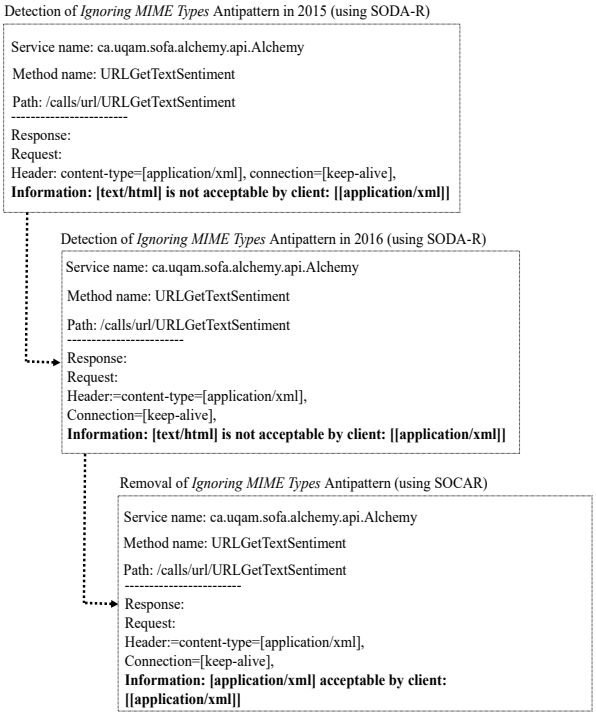


Fig. 6: Evolution of *Ignoring MIME Types* REST Antipattern.

demands content type in the XML format and the server has the content type in JSON format, then SOCAR must add the content type XML dynamically into the response to fulfill the client's request. SOCAR analyses HTTP requests based on the definitions of antipatterns and the corrections suggested in Table 6.

5.3 Corrections for Forgetting Hypermedia Antipattern

Algorithm 2 shows the algorithm used for the correction of the *Forgetting Hypermedia* antipattern, which occurs in the absence of URL links in the HTTP response and restricts the usability of the REST APIs. Hypermedia is the concept of linking resources, i.e., a set of connected resources when applications move from one state to another [59]. The links can point to HTML pages, files, or images [56], on which it is safe for the clients to fall back. API developers design and rely on resource URIs but the client might not receive and follow such links because the server never exposes them in its responses. The correction of the *Forgetting Hypermedia* antipattern also does not work if the server provides status code in the 4xx and 5xx series, i.e., the absence of URL links does not matter if the service is temporarily unavailable. Ideally, REST developers must provide at least one URL link to avoid this antipattern.

Client developers can ask for resources or links in various forms, including JSON, XML, or HTML pages. The responses provided by the server may combine certain attributes like meta-data or location of links. While processing the response, client developers check each link and its resource types. The evolution of this antipattern shows the

changes in responses in terms of changes in *location* attribute or formats in the links provided by the API providers.

5.4 Results

We now discuss the results after applying the SOCAR recommended corrections for the eight REST antipatterns on 11 REST APIs. For each REST API and analysed URI, we report (1) the Precision (P) as the total number of validated true antipatterns with respect to the total number of corrected antipatterns, i.e., the number of correction recommendations that are correct and (2) the Recall (R) as the total number of corrected true antipatterns with respect to the total number of existing true antipatterns, i.e., the cases where SOCAR recommended a correction but should not or should have recommended another correction.

We manually validated every occurrence of the antipatterns. The manual validation for the correction of true antipatterns of 204 URIs is effort-prone. Therefore, we choose a sample size of 134 (70% of the URIs) from 2014 and 2020 to calculate the average precision and recall of SOCAR.

Table 7 reports the URIs analysed for each REST API on which we applied SOCAR for the correction of antipatterns for 2014, 2017, and 2020, respectively. It also reports the numbers of each corrected antipatterns by SOCAR based on the selected sample data set. Tables 8 and 9 report the total numbers of antipattern detected by SODA-R and corrected by SOCAR. Detection(D) ration with respect to the correction ratio is also reported. SOCAR report an average precision of 75.90% and recall of 67.72%based on manual validation.

SOCAR corrected most of the reported antipatterns based on the operations reported in Table 6. For example, we collected the correction approach of *Ignoring Status Code* antipattern from Bitly and Dropbox. Similarly, the Charlie Harvey API reports the removal of *Ignoring MIME Types* antipattern instances. We did not detect any instances of *Tunneling through GET* antipattern in Facebook and YouTube that helps to refine the correction definition of antipatterns reported in [59]. We could not find any relative change in the tunneling antipatterns. The total number of instances for tunneling antipattern for Facebook and YouTube remain the same for 2017. In contrast, for Music Graph, Bitly in the 2014 and 2020, the total number of instances for tunneling antipatterns increased. We studied the trace histories of Facebook, Ohloh, and YouTube for the corrections of the Tunneling antipatterns and implemented their corrections in SOCAR.

SOCAR performs the correction for antipatterns by adding missing attributes. For example, the *Ignoring MIME Types* antipattern is based on a single representation of a MIME type. If some resources have a single representation, SOCAR adds other MIME types after analysing the response dynamically. For example, we add one more resource representation, like JSON, if only the XML were used. However, we do not have physical access to the resources of REST APIs providers, for which we only can get data from the response and add a resource representation. Then, we show this to the client if asked different resource format. For example, if a client of Facebook API requests MIME type in XML format but the Facebook API only returns MIME type in JSON, it is directed to the *Ignoring MIME Types* antipattern.

SOCAR corrects all available antipatterns, the results of which are reported in Table 6. As reported in Table 6, the correction heuristics used by SOCAR increases the instances of the *Entity Linking*, *Response Caching*, and *Content Negotiation* design patterns. The complete trace histories of StackExchange and YouTube also help in knowing the best practices used by prominent API providers, and, for example, they helped us to remove the 12 instances of the FH antipattern in Dropbox. The recommendations by academia and industry are also helpful with the use of HTTP methods GET and POST. Some of the API providers changed the methods used to call some of their URIs, e.g., from GET to POST, which are also reflected in our tool.

The instances of IMT antipattern are not corrected by SOCAR for Facebook and Dropbox. Bitly provide IMT as XML too, but SOCAR cannot parse the response as per the client request. Moreover, some of the antipatterns like BSD are also not corrected because REST API providers continuously change their changelog and tool need to be updated for each URI as per the information updated by the REST API providers. SOCAR must update its catalog of standardised headers for few URIs, updated in 2020.

6 Validation

We now validate the corrections recommended by SOCAR with the help of professionals actively involved in developing REST APIs. We use a questionnaire for this validation. We discuss in the following the data gathering process, the online survey and professional developers' forums, the developers' feedback, and the results of the validation.

6.1 Study Design

We prepared an online questionnaire[9] to gather and analyse the developers' opinion on REST antipatterns. The questionnaire mainly focuses on developers' needs and understanding of the use of different practices to design REST APIs, as reported in Table 4. Therefore, the questionnaire seeks opinion on whether the professional developers support the proposed definitions while we discuss threats to its validity in Section 7.4. The final questionnaire contains:

1) An example of each antipattern and its recommended correction along with an example of the applied correction;
2) A question about the correction definition and its applied correction to collect whether participants agree with them;
3) Participants were asked to provide a proposed solution if they did not agree with the proposed recommendation.

We choose the examples by following the guidelines presented in Murphy et al.'s book [62]. The survey provides four key components: (1) problem definition of each antipattern; (2) examples associated to each antipattern; (3) known implementations in the industry; and, (4) correction rules implemented in SOCAR. Results are collected and available online[1] with detailed information.

[9]https://www.surveymonkey.com/r/7VF7NR5

TABLE 7: Corrections for the Eight REST Antipatterns.

API Names	URIs	BSD	FH	IMT	ISC	IC	MC	TTG	TTP	*Precision(P)*	*Recall (R)*
Alchemy	9	9	1	2	1	7	0	4	5	29/29	29/29
Bitly	15	14	0	14	0	0	0	2	2	30/32	18/32
Charlie Harvey	12	12	0	0	0	0	0	0	0	12/12	12/12
Dropbox	17	17	14	0	0	8	0	2	4	43/45	31/45
Facebook	21	21	21	21	0	4	0	0	0	46/67	46/67
StackExchange	51	24	0	51	0	0	0	0	0	24/75	22/75
YouTube	17	12	0	14	0	3	0	0	0	12/29	14/29
										Avg(P) 75.90%	**Avg(R) 67.72%**

TABLE 8: Antipattern Corrections by SOCAR for 2017.

API Names	URIs Analysed	Antipatterns Detected(D)	Antipatterns Corrected(C)	D/C ratio
Alchemy	9	31	31	100%
Bitly	15	24	24	100%
Charlie Harvey	12	12	12	100%
Dropbox	17	45	45	100%
Facebook	21	64	64	100%
MusicGraph	19	40	40	100%
Ohlo	7	7	7	100%
StackExchange	150	231	231	100%
Twitter	24	78	78	100%
YouTube	14	37	37	100%
Zappos	9	10	10	100%
Total	**297**	**579**	**579**	
Detected and Corrected by as per SOCAR recommendations 100%				

TABLE 9: Antipattern Corrections by SOCAR for 2020.

API Names	URIs Analysed	Antipatterns Detected (D)	Antipatterns Corrected (C)	D/C ratio
Bitly	10	36	36	100%
Charlie Harvey	12	12	12	100%
Dropbox	5	23	23	100%
Facebook	21	71	71	100%
StackExchange	150	231	231	100%
Twitter	24	22	22	100%
YouTube	14	22	22	100%
Zappos	9	10	10	100%
Total	**245**	**427**	**427**	
Detected and Corrected by as per SOCAR recommendations 100%				

6.2 Participants

We invited practitioners, e.g., API developers, of major REST API providers working with REST APIs as well as students who use, develop, or research REST APIs. We invited these practitioners by email. We also shared a link to our questionnaire on known developers forums and on social media platforms used by the Facebook[10], LinkedIn[11], and Twitter[12] and even in YouTube, e.g., used by Zappos and Twitter APIs to reach their developers' community.

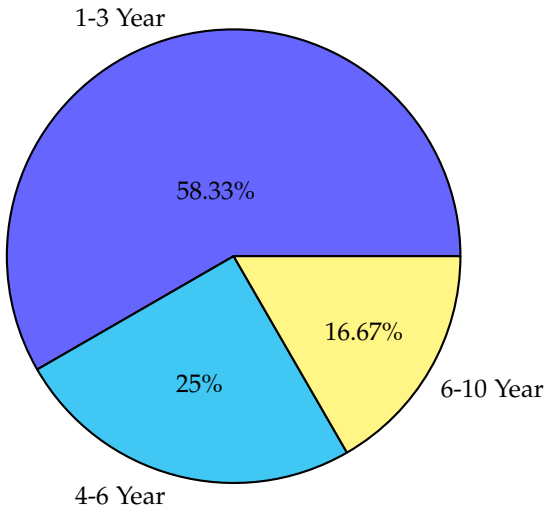
In total, 24 participants completed our questionnaire. Out of these 24 participants, two are graduate students and 22 are professional REST developers. The survey participants were also asked to provide their working experience on software development and REST APIs as depicted in Figure 7. The complete information on participants and their responses are also available on our Web site[1].Most of the participants have experience less then 4 years and only four of the participant have experience grater then 6 years.

[10]https://developers.facebook.com/community/

[11]https://www.linkedin.com/company/project-developer-forum-ltd

[12]https://developer.twitter.com/en/community

Fig. 7: Participants' Experience with REST APIs.



6.3 Results

TABLE 10: Evaluation of the SOCAR based on the Survey.

Antipatterns	Agreed	Not Agreed	Agreement Ratio %
ISC	23	1	95.83%
MC	20	4	83.33%
BSD	22	2	91.66%
FH	23	1	95.83%
IC	22	2	91.66%
TTP	21	3	87.50%
TTG	22	2	91.66%
IMT	22	2	91.66%
Average Agreement Ratio			91.12%

Table 10 reports the average agreement ratio on REST antipatterns correction definitions based on the questionnaire. The results of the questionnaire are also available online[1].

Four professional developers disagreed with the correction proposed for *Misusing Cookies* antipattern. They do not agree with the idea of removing the *set-cookie* parameters.

Two participants disagreed with our proposed correction for Breaking Self-Descriptiveness. The reason for using a standardised header may force the professional developer to focus on a specific list of headers. This list can be updated and the client developers need to update the code too. This increases the maintenance cost associated with the API.

Two of the participants also reported not using "cache-Control" as "no-cache". These results match Observation 2, as the use of *Ignoring Caching* post has more than 43 thou-

sand views on the Stack Overflow. The developers argued to use the values "private" or "public" with a maximum age and a specific ETag value.

Four of the Professional developers also did not agree on the definition of *Misusing Cookies* as antipatterns to avoid using *cookies* in session state. Cookies are used by most of the major REST API providers by showing the message based on the the opinion to accept *Cookies*, but this is recommended to be used on client system for specific time interval as per standard guidelines [60].

Two of the participants also prefer to use *GET method* for *Tunneling Through GET* for all sort of actions like creating ,deleting and updating resources. However, for the definition of *Tunneling Through Post*, three of developers prefer to use *POST* method for server side request for updating and retrieving resources.

Two professional developers from LinkedIn, YouTube, and Facebook also did not agree on the use of multiple MIME types. We get a 91.12% agreement ratio for *SOCAR* approach based on the results of survey participants. The results of the case study provide developers views about recommended strategies for the correction of REST APIs. Their feedback also provides evidence that professional developers recommendations are handled accurately for the development of SOCAR.

7 Discussions

In this section, we further discuss our detection, recommendations for correction, results of their applications, and the threats to the validity of our results. The following sections focus on Alchemy, Bitly and YouTube for the following reasons:

- Some of the APIs do not make available multiple version history and provide only a single version running at a time, like Alchemy and YouTube. If multiple version history is not available, we checked the online change-logs of the APIs, for example, for Alchemy, Bitly and YouTube.
- We also want to check whether there is a change in antipattern instances, especially during a migration phase or in a new release of some APIs. For example, the Alchemy API migrated in 2017 from Alchemy to IBM.
- YouTube has a single version running in the last three years. The change-log and trace history of YouTube help to understand the evolution of antipatterns from 2015 to 2020.

7.1 About the Quantitative Study

The quantitative study, and subsequently the SOCAR approach, depends on the validity and relevancy of the REST antipatterns. We were interested in observing whether or not the antipatterns defined in the SOFA framework (SODA-R) were still relevant and present in recent (versions of) REST APIs.

Our observational study, through Observation 1, shows that the eight considered REST antipatterns are still present in recent versions of REST APIs, as shown in Table 3, in particular in the rows corresponding to the year 2020.

Yet, the presence of occurrences of these REST antipatterns does not mean, *per se*, that they are still relevant because developers' practices have evolved. Observation 2 and Table 4 show that, for the year 2020, StackOverflow contains questions/answers regarding these REST antipatterns, which seems to show that they are still relevant today.

We did not perform a search for new REST antipatterns or for updated versions of existing REST antipatterns. For example, the *Ignoring MIME Type* antipattern may be currently becoming obsolete with the advent of JSON and could be deprecated in favour of a new *Non-JSON MIME Type* antipattern "enforcing" the use of JSON as good practice.

Observations 1 to 3 show that the REST antipatterns considered in this study were still relevant in 2020. Consequently, we leave for future work the identification, definition, and study of novel REST antipatterns.

7.2 About the Qualitative Study

We validate our correction recommendations with the help of practitioners who have experiences in developing REST APIs. In particular, practitioners had the highest concern regarding the *Ignoring MIME Types* antipattern. Practitioners prefer to use a single resource presentation type, JSON. This dependence on a single resource representation reduces the accuracy of SOCAR for the correction of the *Ignoring MIME Types* antipattern because two of the practitioners did not consider multiple MIME types important. They also suggested that client developers should rely on the MIME types set by the server. They also raised an issue on using multiple MIME types in multiple formats, as it burdens server-side developers. We can interpret this as: some antipatterns are actually *not* antipattern in practice because developers balance guidelines and costs to implement those guidelines.

The survey participants also suggested using caching. However, they did not agree on the recommendation to set the *cache-control* attribute to `public` or `private`. The *cache-control* attribute used in SOCAR comes from the literature [59]. We also found a difference of opinions for *Tunneling through GET/POST* antipattern, which suggests using HTTP GET request for accessing resources and not to modify them. Respondents of the survey also left this option open for API providers and suggested forcing client developers to use tunneling options based on what is provided by the servers.

7.3 About the SOCAR Approach

We presented the SOCAR approach to recommend some corrections or apply other corrections to remove occurrences of REST antipatterns in REST APIs. We build our approach on the assumption that we do not have access to the actual source code of the REST APIs. SOCAR either recommend corrections that could be implemented by the REST API developers or perform corrections on responses to protect clients from the negative impact of these antipatterns.

Hence, SOCAR is really a "shield" against the negative impact of REST antipatterns for REST API clients and, when a correction is not feasible on the clients' sides, a recommendation system to support REST API developers in removing some REST antipatterns.

SOCAR is but a first step towards the systematic detection and correction of REST antipatterns in REST API on the providers' side. It shows that it is possible to identify relevant corrections from the detection and evolution of REST antipatterns and from the community discussions. It also shows that some REST antipatterns can even be corrected on the clients' sides.

We did not apply SOCAR on open-source REST APIs whose source code is available to correct concretely because we wanted first to assess whether correcting/recommending corrections to REST antipatterns was feasible and with good precision 75.90% and recall 67.72%. Section 6 showed that developers mostly agree (at 91.2%) with the recommendations of SOCAR.

Based on SOCAR, we humbly believe that the community interested in improving the quality of REST APIs could continue collecting corrections to the REST antipatterns studied in this paper and to other antipatterns. It could then propose corrections to these antipatterns on both the clients' and providers' sides (when possible). It could also curate a list of open-source REST APIs and use some of these APIs to validate the corrections on the providers' sides.

7.4 Threats to Validity

To ensure the reproducibility of our study, we share the requests, responses, developers' answers along with all the definitions of the antipatterns on our Web site[1]. The survey is also available online[9]. The accuracy of SOCAR depends on the trace histories and the accuracy of SODA-R when detecting antipatterns.

Construct Validity: Threats related to *construct validity* concern the relationship between theory and observations and the accuracy of the experiment performed to answer our research questions. To minimise this threat globally, we applied a mixed-method analysis, i.e., a combination of *quantitative* and *qualitative* study.

Quantitatively, we analysed the evolution of 11 REST APIs by performing the detection of REST antipatterns. *Qualitatively*, we analysed developers' forums and reported that developers are concerned with the evolution and quality of their REST APIs.

We relied on SODA-R [12] for the detection of antipatterns and for the extraction of traces, which was already used in 2014 to study traces and antipattern evolution. It has an average precision of 89.4% and an average recall of 94% for the detection of REST antipatterns.

Construct validity may also be negatively impacted by the design and administering of our survey, which may have led participants to agree with our statements for simplicity and desirability. We must accept this threat and future work should include a more sound survey, e.g., using Likert scales, send to more participants.

Internal Validity: Threats to *internal validity* concern the factors that may affect our results. We based the definition of the antipatterns and their corrections on the literature, both from the academic and developers' forums.

We also mitigated this threat by asking developers' opinions about alternative corrections for each antipattern. However, we understand but must accept that developers' opinions are subjective and vary with their experiences and other factors beyond our control.

We use the most relevant tag to extract the developer opinion from Stack Overflow. There might be a possibility that some of the problems are also posted with other tags. We tried to mitigate this threat by studying the latest question or posts from StackOverflow about REST [72].

In addition, SOCAR allows users full freedom to change the correction definitions and–or define their own corrections for specific antipatterns.

External Validity: Threats to *external validity* concern the generalisability of the validation results. We recommended corrections for all instances of the detected REST antipatterns and implemented real-time corrections for common APIs like Facebook, StackExchange, or YouTube.

SOCAR corrections are based on calls to URIs. Changes to the URIs or their responses may affect the number of instances corrected for each REST antipattern. REST API providers information might also affect the results and accuracy of SOCAR. We tried to mitigate this threat by providing guidelines to run the tool, available on our Web site[1].

8 Conclusion and Future Work

Application Programming Interfaces (APIs) are the software, programmatic interfaces that help applications and databases share different functionalities and data [1]. Web giants, like Facebook, Google, Twitter, or YouTube, provide REST APIs to provide access to their resources [2]. Poor design practices, i.e., *antipatterns* (as opposed to *design patterns*) could be introduced due to API evolution to meet the needs of both service providers and consumers.

We applied a mixed-method analysis to understand the evolution of REST APIs and REST antipatterns. *Quantitatively*, we analysed the evolution of 11 REST APIs, including Facebook, Twitter, and YouTube over five years by computing changes in request/response data and occurrences of eight REST antipatterns. Through the quantitative study, we showed that (1) REST APIs and REST antipatterns evolve over time. We noticed a high relative change for antipatterns in APIs from the year 2015 until 2020. For example, there is an increase in *Ignoring MIME Types* antipatterns for most of the APIs since API providers now intend to rely on a single resource representation, e.g., JSON. *Qualitatively*, we analysed developers' forums and reported that developers are concerned with the evolution and quality of REST APIs. Numerous issues related to poor and best REST design and development practices are reported on professional developer forums. Thus, there is a need to tackle REST antipatterns and, thus, assist developers in designing APIs free of antipatterns.

We developed a tooled approach, SOCAR (Service Oriented Correction of Antipatterns in REST), to recommend corrections to REST antipatterns, and, where applicable, remove REST antipatterns in REST APIs. We manually validated our results based on the instances of detected antipatterns as compared to the corrected antipatterns. We also conducted an online survey to mitigate the threat of our approach and know the opinion of the professional REST API developers. Most participants agreed on the definitions of *Ignoring Status Code* and *Forgetting Hyper Media* antipatterns. Survey results are promising with an agreement ratio of 91.12%. The opinion from the practitioners and academics

involved in REST development helped us to improve the correction of REST antipatterns in SOCAR with an average precision of 75.90% and recall of 67.72%.

Work in progress includes mining the trace history of each change in some REST APIs and create a repository for these changes. Another work in progress is a study of the problems faced by client APIs.

Future work includes surveying developers about REST antipatterns. We want to investigate the number of changes for each REST URI and analyse which APIs survived for a longer period. We will use collections of errors reported for specific REST APIs to correlate changes with antipatterns. Using developers' feedback, we plan to improve our correction algorithms to fit developers' expectations. We will also to add an automated feedback recommendation and correction system that helps developers to design REST APIs free from antipatterns and refactor antipatterns automatically. We are also interested in reporting a benchmark for HTTP status codes and descriptions after collecting data from different API providers. Future works includes also identifying novel REST antipatterns, relevant to recent developers' practices.

ACKNOWLEDGMENT

The authors thank the professional REST developers who answered our survey. We also thank the many REST developers who asked and–or answered questions publicly on developers' forums. We are grateful to the graduate students who manually validated REST APIs requests and responses. This work was supported by the International Research Support Initiative Program (IRSIP) and funded by the Higher Education Commission (HEC), Pakistan. It was also partly funded by the Canada Research Chair program.

REFERENCES

[1] D. Jacobson, D. Woods, and G. Brail, *APIs: A strategy guide*. " O'Reilly Media, Inc.", 2011.

[2] D. Bermbach and E. Wittern, "Benchmarking web API quality," in *16th International Conference on Web Engineering (ICWE)*. Springer, 2016, pp. 188–206.

[3] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Investigating Web APIs on the World Wide Web," in *8th IEEE European Conference on Web Services*. IEEE, 2010, pp. 107–114.

[4] D. Renzel, P. Schlebusch, and R. Klamma, "Today's top" RESTful" services and why they are not RESTful," in *13th international conference on Web Information Systems Engineering*, 2012, pp. 354–367.

[5] F. Bülthoff and M. Maleshkova, "RESTful or RESTless–Current state of today's top Web APIs," in *11th International Conference on the Semantic Web*. Springer, 2014, pp. 64–74.

[6] J. Kopeckỳ, P. Fremantle, and R. Boakes, "A history and future of Web APIs," *it-Information Technology*, vol. 56, no. 3, pp. 90–97, 2014.

[7] A. Demange, N. Moha, and G. Tremblay, "Detection of SOA Patterns," in *11th International Conference on Service-Oriented Computing(ICSOC)*. Springer, 2013, pp. 114–130.

[8] T. Erl, *SOA Design Patterns* . Pearson Education, 2008.

[9] T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian, *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*. Prentice Hall Press, 2012.

[10] D. Bán and R. Ferenc, "Recognizing antipatterns and analyzing their effects on software maintainability," in *14th International Conference on Computational Science and Its Applications(ICCSA)*. Springer, 2014, pp. 337–352.

[11] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *15th European conference on Software maintenance and reengineering (CSMR)*. IEEE, 2011, pp. 181–190.

[12] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of REST patterns and antipatterns: a heuristics-based approach," in *12th International Conference on Service-Oriented Computing(ICSOC)*. Springer, 2014, pp. 230–244.

[13] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, "Are RESTful APIs well-designed? Detection of their linguistic (anti) patterns," in *13th International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2015, pp. 171–187.

[14] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Towards a REST Cloud Computing Lexicon," in *7th International Conference on Cloud Computing and Services Science (CLOSER)*, 2017.

[15] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are REST APIs for cloud computing well-designed? An exploratory study," in *13th International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2016, pp. 157–170.

[16] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[17] L. Li and W. Chou, "Design and describe REST API without violating REST: A Petri net based approach," in *9th IEEE International Conference on Web Services (ICWS)*. IEEE, 2011, pp. 508–515.

[18] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "REST APIs: a large-scale analysis of compliance with principles and best practices," in *16th International Conference on Web Engineering (ICWE)*. Springer, 2016, pp. 21–39.

[19] F. Haupt, F. Leymann, A. Scherer, and K. Vukojevic-Haupt, "A Framework for the Structural Analysis of REST APIs," in *IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2017, pp. 55–58.

[20] M. Athanasopoulos and K. Kontogiannis, "Extracting REST resource models from procedure-oriented service interfaces," *Journal of Systems and Software*, vol. 100, pp. 149–166, 2015.

[21] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns," *International Journal of Cooperative Information Systems*, p. 1742001, 2017.

[22] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009, pp. 75–84.

[23] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Ieee, 2011, pp. 125–134.

[24] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *5th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2001, pp. 30–38.

[25] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[26] C. Mills, E. Parra, J. Pantiuchina, G. Bavota, and S. Haiduc, "On the relationship between bug reports and queries for text retrieval-based bug localization," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3086–3127, 2020.

[27] A. Rathee and J. K. Chhabra, "Mining Reusable Software Components from Object-Oriented Source Code using Discrete PSO and Modeling Them as Java Beans," *Information Systems Frontiers*, pp. 1–19, 2019.

[28] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[29] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.

[30] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "A recommendation system for repairing violations detected by static architecture conformance checking," *Software: Practice and Experience*, vol. 45, no. 3, pp. 315–342, 2015.

[31] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19–36, 2018.

[32] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, 2016.

[33] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)," *IEEE Transactions on Software Engineering*, 2017.

[34] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, 2013.

[35] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 25–34.

[36] T. Mens, G. Taentzer, and O. Runge, "Analysing refactoring dependencies using graph transformation," *Software and Systems Modeling*, vol. 6, no. 3, p. 269, 2007.

[37] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, "RefBot: Intelligent Software Refactoring Bot," in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 823–834.

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

[39] J. Purushothaman, *RESTful Java Web Services*. Packt Publishing Ltd, 2015.

[40] J. Sandoval, *Restful java web services: Master core rest concepts and create restful web services in java*. Packt Publishing Ltd, 2009.

[41] S. Allamaraju, *Restful web services cookbook: solutions for improving scalability and simplicity*. " O'Reilly Media, Inc.", 2010.

[42] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[43] T. Espinha, A. Zaidman, and H.-G. Gross, "Web API growing pains: Loosely coupled yet strongly tied," *Journal of Systems and Software*, vol. 100, pp. 27–43, 2015.

[44] R. Daigneau, *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley, 2011.

[45] C. Mateos, M. Crasso, A. Zunino, and J. L. O. Coscia, "Revising wsdl documents: Why and how, part 2," *IEEE Internet Computing*, vol. 17, no. 5, pp. 46–53, 2013.

[46] C. Mateos, J. M. Rodriguez, and A. Zunino, "A tool to improve code-first web services discoverability through text mining techniques," *Software: Practice and Experience*, vol. 45, no. 7, pp. 925–948, 2015.

[47] J. L. Ordiales Coscia, C. Mateos, M. Crasso, and A. Zunino, "Anti-pattern free code-first web services for state-of-the-art Java WSDL generation tools," *International Journal of Web and Grid Services*, vol. 9, no. 2, pp. 107–126, 2013.

[48] G. Salvatierra, C. Mateos, M. Crasso, and A. Zunino, "Towards a computer assisted approach for migrating legacy systems to SOA," *Computational Science and Its Applications(ICCSA) 2012*, pp. 484–497, 2012.

[49] J. L. O. Coscia, C. Mateos, M. Crasso, and A. Zunino, "Refactoring code-first web services for early avoiding wsdl anti-patterns: Approach and comprehensive assessment," *Science of Computer Programming*, vol. 89, pp. 374–407, 2014.

[50] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Discoverability anti-patterns: frequent ways of making undiscoverable web service descriptions," in *Proceedings of the 10th Argentine Symposium on Software Engineering (ASSE)*, 2009, pp. 1–15.

[51] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, "Revising WSDL documents: Why and how," *IEEE Internet Computing*, vol. 14, no. 5, pp. 48–56, 2010.

[52] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinnéide, "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, vol. 10, no. 4, pp. 603–617, 2015.

[53] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, pp. 18–39, 2015.

[54] H. Wang and M. Kessentini, "Improving Web Services Design Quality Via Dimensionality Reduction," in *15th International Conference on Service-Oriented Computing*. Springer, 2017, pp. 499–507.

[55] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures," *Software: Practice and Experience*, vol. 42, no. 5, pp. 559–583, May 2012.

[56] F. . Reschke, "Hyper Text Transfer Protocol." [Online]. Available: https://tools.ietf.org/html/rfc7230#section-6.1

[57] U. Souichi, "Information processing apparatus and method of acquiring trace log," June 2014, uS Patent 8,745,595.

[58] M. Terpolilli, "Trace log rule parsing," Mar. 26 2013, uS Patent 8,407,673.

[59] S. Tilkov, "Rest anti-patterns." [Online]. Available: https://www.infoq.com/articles/rest-anti-patterns

[60] Saint-Andre, "Best current practises." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6648

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

[62] L. Murphy, T. Alliyu, A. Macvean, M. B. Kery, and B. A. Myers, "Preliminary Analysis of REST API Style Guidelines," *Ann Arbor*, vol. 1001, p. 48109, 2017.

[63] S. Sohan, F. Maurer, C. Anslow, and M. P. Robillard, "A study of the effectiveness of usage examples in REST API documentation," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 53–61.

[64] E. Ozdemir, "A General Overview of RESTful Web Services," *Applications and Approaches to Object-Oriented Software Design: Emerging Research and Opportunities*, pp. 133–165, 2020.

[65] B. De, "Api management," in *API Management*. Springer, 2017, pp. 15–28.

[66] M. S. Faisal, A. Daud, A. U. Akram, R. A. Abbasi, N. R. Aljohani, and I. Mehmood, "Expert ranking techniques for online rated forums," *Computers in Human Behavior*, vol. 100, pp. 168–176, 2019.

[67] L. Guerrouj, S. Azad, and P. C. Rigby, "The influence of app churn on app success and stackoverflow discussions," in *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 321–330.

[68] P. Arora, D. Ganguly, and G. J. Jones, "The good, the bad and their kins: Identifying questions with negative scores in stackoverflow," in *7th IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 2015, pp. 1232–1239.

[69] A. Wingkvist and M. Ericsson, "Asked and answered: Communication patterns of experts on an online forum," in *36th Information Systems Research Seminar in Scandinavia*, 2013.

[70] Roy-Fielding, "Best current practises." [Online]. Available: https://https://datatracker.ietf.org/doc/html/rfc2616

[71] M. Nottingham, "Best current practises." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6585

[72] J. Au-Yeung, "Best practices for rest api design." [Online]. Available: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/

[73] F. Palma, N. Moha, and Y. Guéhéneuc, "UniDoSA: The Unified Specification and Detection of Service Antipatterns," *IEEE Transactions on Software Engineering*, vol. 45, no. 10, pp. 1024–1053, 2019.

[74] B. Walter and T. Alkhaeir, "The relationship between design patterns and code smells: An exploratory study," *Information and Software Technology*, vol. 74, pp. 127–142, 2016.