Exploratory Analysis of the Structural Evolution of public REST APIs

Nicolás Robles[1], Nicolás Potes[1], Kelly Garcés[1], Javier Luis Canovas Izquierdo[2], Jordi Cabot[2]

[1]Department of Systems and Computing Engineering
Universidad de los Andes, Colombia

[2]IN3 - UOC; ICREA, Spain

{nr.robles, n.potes, kj.garces971}@uniandes.edu.co

jcanovasi@uoc.edu, jcabot@icrea.cat

Abstract. *The software industry is in continuous evolution, forcing developers to quickly adapt to new requirements to catch up with the latest trends. A clear example is the huge demand for web-based APIs to connect all kinds of services among them. Like any other piece of software, a Web API is continuously changing, and with each change, all client applications must evolve. This adaptation process is critical and essential for software developers. In this paper, we present an exploratory evaluation of the most common changes occurring during the evolution of REST APIs. We define a taxonomy of structural API changes, which we classify according to their impact on client-side software; and propose a repository mining process to identify these changes in real Web APIs. We apply this process to a large set of Azure APIs from* APIsGuru*, a well-known Open Source API repository. Based on the analyzed dataset from* APIsGuru*, we found that breaking changes tend to decrease when a new version of an API from this dataset is released. Other useful findings and insights are discussed throughout the article.*

1. Introduction

The digital transformation has created a world where services and data are accessed through the Internet; and therefore service providers make their data available through Application Programming Interfaces (APIs). APIs can be classified as *libraries* or *remote APIs* considering the place where they are hosted. The former runs in the same environment as the client software, whereas the latter is hosted in a different environment and reached by the client application using remote access protocols. When the protocol is HTTP, the APIs are called Web APIs. This work targets the evolution of REST Web APIs which are the most common type of Web API used today [Halili and Ramadani 2018]. It is worth mentioning that REST (REpresentational State Transfer) is an architectural style that guides the design and development of Web APIs.

Like any other piece of software, service providers need to evolve their APIs, which frequently implies the introduction of changes in the released API versions. As a result, developers need to adapt (co-evolve) their software to support the new API version, which requires full knowledge of API changes and their impact. [Li et al. 2013] note that updating Web API clients is a more critical task than updating library clients because

the former generally can not keep using the old API version (as it may not be available anymore).

Change detection between REST API versions is generally addressed by manual reviewing the API documentation and perusing the API controllers used on a client application to find the occurrences of evolving methods [Koçi et al. 2019, Wang et al. 2014]. However, these approaches are time-consuming and dependent on the API's implementation. This situation has motivated us to mine a public REST API repository to expand the body of knowledge about Web API evolution. By knowing better how the API evolution happens in practice, developers could be adequately prepared for it, both in terms of determining what types of changes they should be prepared for and in terms of how impactful those changes risk to be for the client applications consuming the APIs.

We mined a large set of APIs by Azure available at APIsGuru[1], a well-known API repository, to better understand the most common changes occurring in the evolution of REST APIs. Each change was classified either as a *breaking* or *non-breaking* change. A breaking (BR) change means there is an impact on the API client applications that damages their functionality. In turn, a non-breaking (NON-BR) change preserves the compatibility with previous API versions, and thus, the client code will keep running [Dig and Johnson 2006]. Understanding the frequency and type of changes is therefore key to prioritize and plan any API evolution process. The importance of this problem appeared often in our discussions with practitioners working in the development and maintenance of APIs.

Besides this characterization of API changes, this paper also contributes: (i) a taxonomy of possible REST API changes classified by their impact on client software; (ii) a tool to identify API changes and (iii) a mining process to (semi) automate the extraction and analysis of APIs in an API repository. We use OpenAPI, a standard to describe REST APIs in a language-agnostic way, to analyze and compare REST APIs. Thus, our approach does not require any access to API source code and therefore is independent of the programming language the API is implemented with. Note that there are ways to generate the OpenAPI specification out of API implementation if needed [Cao et al. 2017].

The article is organized as follows. Sections 2, 3 and 4 present the research questions, method and results, respectively. Sections 5 and 6 present the validation and the threats to validity. Section 7 reports the related work, while Section 8 ends the paper and presents the future work.

2. Research Questions

This section states the three research questions to guide our understanding of REST APIs evolution:

RQ1 How frequent are breaking changes in API evolution?
RQ2 What are the most frequent types of changes introduced in API evolution?
RQ3 Is it possible to get insights on the stability of an API by reviewing its evolution process?

These questions can be responded at two different levels of granularity: *repository level* and *API level*. *Repository level* refers to when the analysis is based on all the APIs of

[1] https://apis.guru/

the dataset (general), on the other hand, *API level* refers to when the analysis is based on certain APIs (particular). We answer RQ1 and RQ2 at repository level to provide overall information on the evolution of the analyzed APIs. Researchers could take advantage of the findings to develop techniques and tools to ease the adaptation of client programs to frequent changes. On the other hand, R3 will be addressed at API level to compare the stability of APIs and get insights about its quality (i.e., the fewer changes, the more stable the API, meaning the less potential impact on client software).

We reached these questions and the granularity level based on our own experience and the opinion of a group of senior software developers with whom we had informal meetings during this research.

3. Research method

The research method we developed to answer our research questions covers the following steps: (1) definition of a taxonomy of API changes, (2) development of a tool to mine changes in evolving APIs, (3) execution of the mining process. The subsequent sections describe each step in detail.

3.1. Taxonomy of API changes

To build our taxonomy of API changes, we followed a process composed of three phases: (1) kick-start the taxonomy by analyzing existing proposals; (2) manual review of existing Web APIs to identify new change types; and (3) taxonomy consolidation.

Our taxonomy is inspired by the proposal made by [Li et al. 2013], which we extended with the following types of changes: i) add path; ii) change parameter lower bound; iii) relocate parameter; and iv) change default parameter value. We identified these new categories after manual reviewing twenty OPENAPI specifications available on the APISGURU repository. To ensure high representativity, each reviewed API satisfies three criteria: it has at least two versions, it includes at least ten endpoints, and it has had commits recently. We removed the XML Tag type change because we focus on REST APIs which do not use XML [Halili and Ramadani 2018]. Table 1 lists the APIs changes of the taxonomy (see first column). We use the severity column to distinguish between breaking (BR) and non-breaking (NON-BR) changes.

We compared our taxonomy with the ones proposed by [Li et al. 2013], [Wang et al. 2014], [Sohan et al. 2015]. Table 1 shows the results of such comparison (see last four columns). For each API change, the row indicates whether they are included or not in each proposal. As can be seen, our taxonomy covers most of the changes proposed by the aforementioned authors and includes additional ones. It is worth noting that our taxonomy does not cover API changes related to authentication and authorization, as those changes are not identifiable using static analysis and also because the OPENAPI v2.0 lacks elements to represent with accuracy the different types of API authorization.

3.2. API Changes Detection Tool

In this section we present the elements of the tool developed to detect API changes, called API Detection Tool (ADT). We first describe the *Diff Changes Metamodel*, that serves to represent our changes taxonomy and is instantiated by the tool; and then present the ADT tool architecture.We use the Model-Driven Engineering (MDE) approach to create and

CHANGE NAME	SEVERITY	OURS	[Li et al. 2013]	[Wang et al. 2014]	[Sohan et al. 2015]
Change parameter Type	BR	Y	Y	Y	N
Rename parameter	NON-BR	Y	Y	Y	Y
Increase number of parameters	BR	Y	N	Y	Y
Decrease number of parameters	BR	Y	Y	Y	Y
Change return type	BR	Y	Y	Y	Y
Delete path	BR	Y	Y	Y	Y
Change parameter schema type	BR	Y	Y	Y	N
Unsupported request method	BR	Y	Y	N	N
Change default parameter value	NON-BR	Y	N	N	N
Change parameter upper bound	NON-BR	Y	Y	Y	N
Change parameter lower bound	NON-BR	Y	N	Y	N
Change return type schema	BR	Y	Y	Y	N
Relocate parameter	BR	Y	N	N	N
Change consumed type	BR	Y	Y	N	N
Remove restricted access	NON-BR	Y	N	N	Y
Add restricted access	NON-BR	Y	Y	N	Y
Change produced type	BR	Y	Y	Y	N
Rename method	BR	Y	Y	Y	Y
Combine methods	BR	Y	Y	N	N
Split method	BR	Y	Y	N	N
Add path	NON-BR	Y	N	Y	Y
Change auth model	BR	N	N	Y	Y
Change required parameter type	BR	Y	N	Y	N
Add status code	NON-BR	Y	N	N	N
Remove status code	BR	Y	N	N	N

Table 1. Taxonomies comparison.

develop the tool. MDE is a Software development approach which focuses on models, as opposed to source code. Models are built representing different views on a Software system. The main goal is to raise the level of abstraction, and to develop and evolve complex Software systems by manipulating models.

3.2.1. Diff Changes Metamodel

We have designed a metamodel to represent the API changes based on our taxonomy and the metamodel proposed by [Polák and Holubová 2015]. The root concept of our metamodel is *Diff*, which is composed of a set of changes. A change can be *Simple* or *Complex*. A simple change refers to an individual element of a given API undergoing a unique change, either a *Deletion*, an *Addition*, or a *Modification*. Depending on the type of change, a simple change may have references to API elements presented in the former and latter compared versions. A *Complex* change is a set of simple changes that affect multiple API elements. Finally, we represent the changes shown in Table 1 as specializations of the concepts *Simple* and *Complex*.

3.2.2. ADT Architecture

ADT aims to identify changes introduced in a given REST API by comparing two of their versions. The discovery process is performed by a transformation

chain which takes as input two OPENAPI models corresponding to different versions of the same API. These models conform to the already existing *OpenAPI metamodel* [SOM RESEARCH LAB 2018] and are later processed by a comparator that executes EMFCOMPARE and gets a *comparison model*. This comparison model has generic differences (i.e., ADD, DELETE, CHANGE and MOVE), which are processed and transformed to obtain a model conforming to the *Diff changes metamodel* presented before.

3.3. Mining process

We performed a mining process by applying ADT to a large set of OPENAPI definitions by Azure from APISGURU. We selected those APIs having multiple versions and whose definitions conform to OPENAPI v2.0. In the following, we report the main challenges found during the mining process.

OPENAPI models representing OPENAPI definitions were ordered in pairs including the consecutive versions of the API. For example, if an API has three versions, the resulting pairs would be (v1, v2) and (v2, v3). Then, we called ADT for each pair of versions to be compared. We processed the versions in the aforementioned order to ensure the comparisons follow the API evolution path so that the results show the state of the API on each evolutionary step. Finally, we studied the results and generated a set of charts to address our research questions.

4. Results

We have designed a set of visualizations that aims at answering the research questions stated in Section 2 from two different perspectives: i) repository level (overall evolution of analyzed APIs); ii) API level (evolution of selected APIs). Section 5 presents the findings of a survey on the usefulness of the contributed visualizations. Our process mined 225 different APIs, including 1568 different versions, in other words, 784 pairs of versions. Each API reviewed had, on average, 6.97 versions and also on average, 1.72 breaking changes per comparison in each pair of versions. In the following, we take advantage of the visualizations provided to answer the research questions for this dataset. For the sake of space we present the results of RQ1 and RQ2 at repository level, but the same graphics can be generated for a particular API specified by the user. In the section associated to RQ3 we present results at repository and API level because the charts are different at each granularity.

4.1. RQ1: How frequent are breaking changes in API evolution?

An API is considered affected when there is at least one change (breaking or non-breaking). Based on this, the number of affected APIs with respect to the total is: 157 out of 225 studied. That is, 69.78% were affected APIs.

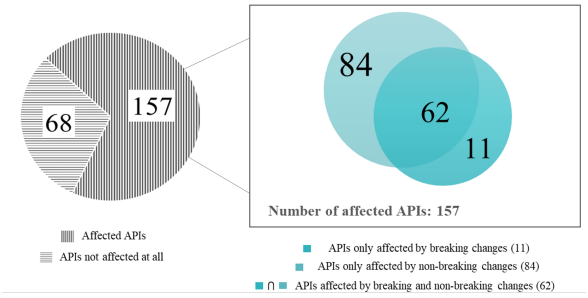

Figure 1. Distribution of APIs by type of change

Figure 1 shows the distribution of APIs by type of change. 11 APIs were affected only by breaking changes, 84 APIs were affected only by non-breaking changes, of these two sets, the intersection was 62 APIs, i.e., APIs affected by both breaking and non-breaking changes. Finally, 68 APIs of the dataset were not affected by any change.

Additionally, we decided to analyze the percentage of pairs of versions that include a breaking change. Based on this, we found that of the 784 pairs of versions analyzed, 102 pairs of versions include at least one breaking change. That is, **13.01%** of the pairs of versions have a breaking change in the studied dataset.

Finally, we performed an analysis by API looking at the percentage of endpoints[2] that changed versus those that did not. In general, for all the APIs in the dataset under study, the following distribution was found:

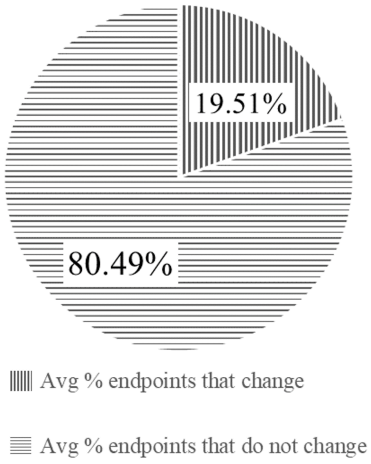

Figure 2. Average % of endpoints that change vs. those that do not)

As can be seen in figure 2, most endpoints of the APIs under study do not tend to change.

[2] An endpoint is a point at which an API connects with a client program

4.2. RQ2: What are the most frequent types of changes introduced in API evolution?

We obtained the following distribution from the dataset of the most popular breaking and non-breaking changes, represented in the figures 3 and 4:

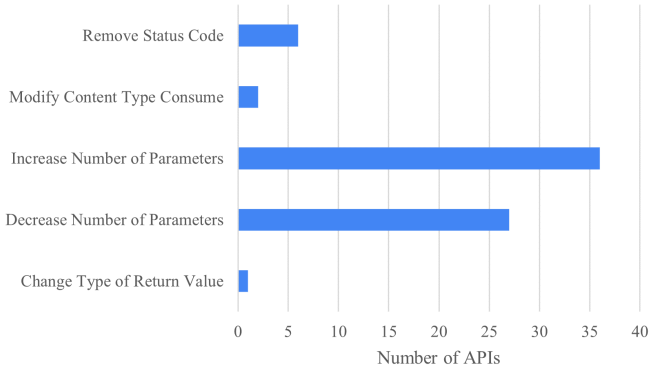

Figure 3. Most popular breaking changes per API

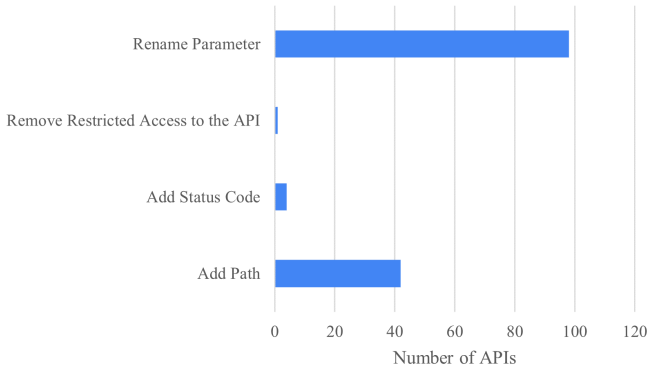

Figure 4. Most popular non-breaking changes per API

- **Breaking change most repeated:** *Increase Number of Parameters* with 658 occurrences
- **Non-breaking change most repeated:** *Rename Parameter* with 1473 occurrences

As can be seen, both changes are very common and are therefore the most frequently repeated. On one hand, the *Increase Number of Parameters* change occurs when the definition of a method is changed by adding more parameters. On the other hand, the *Rename Parameter* change occurs when the name of a parameter is changed.

4.3. RQ3: Is it possible to get insights on the stability of an API by reviewing its evolution process?

To address this research question, we decided to conduct analyses at both repository and API levels.

4.3.1. Repository level

Firstly, to visualize the trend regarding the average age of APIs, measured in the number of versions, figure 5 shows the number of pairs of versions on the x-axis (1-21) and the number of APIs that have *n* pairs of versions on the y-axis . For example, there are 133 APIs that have two pairs of versions, i.e., 4 versions in the dataset.

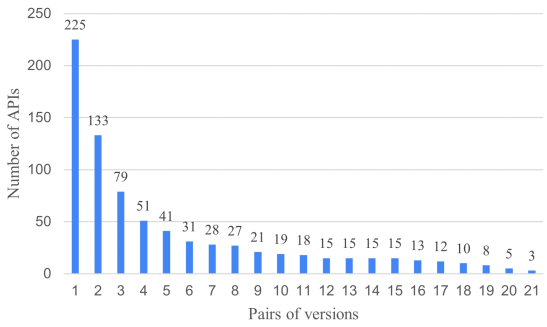

Figure 5. Number of APIs per pair of versions

Based on Figure 5, we can see that the vast majority of APIs have 1 to 4 version pairs in this dataset. Therefore, their average age is around this range.

Secondly, we conducted an analysis of the 225 APIs to find out what was the average disappearance time of breaking changes, measured in versions. To determine that breaking changes are disappearing (decreasing) in APIs, we compared the number of breaking changes found in a given pair of version with that of its immediately previous pair. Based on this, we found that only APIs that have two pairs of versions decrease the breaking changes without increasing them again. Therefore, the average disappearance time for this dataset is **two pairs of versions** (4 versions) To clarify, table 2 exemplifies the above with two concrete APIs: *apimcaches* and *apimdeployment*.

API	PV1	PV2	PV3
apimcaches	10	5	7
apimdeployment	5	2	0

Table 2. Descending breaking changes in APIs' pairs of versions

As shown in table 2, *apimcaches* has 10 breaking changes in its first pair of versions, in its next pair of versions it decreases to 5 breaking changes but finally in its

third pair of versions it increases again its breaking changes (7), therefore, this API is not considered to completely decrease its breaking changes since in some pair of versions it increased again its number of breaking changes or it never decreased them. On the other hand, *apimdeployment* decreases its breaking changes without increasing them again, therefore, this API is considered to have totally decreased its breaking changes.

In addition, we performed an analysis of the evolution of breaking changes for each number of API pairs of versions in the dataset. To this end, we obtained the pairs of versions of each API in the dataset and found that there are APIs that have only one pair of versions up to APIs that have 21 pairs of versions. In addition, we calculated the number of breaking changes for each pair of version. Then, we summed the breaking changes observed in the first pairs of version (i.e., version 1 and 2) of all APIs. We did the same for the second pairs (i.e., version 2 and 3), third pairs and so on. It is worth noting that the number of pairs of versions is variable among the studied APIs. From these aggregated values, we created a Scatter graph that shows the behavior of the breaking changes (y-axis) through the different pairs of versions (x-axis).

Figure 6 shows the pairs of versions on the X-axis and the number of breaking changes per pair of versions on the Y-axis.

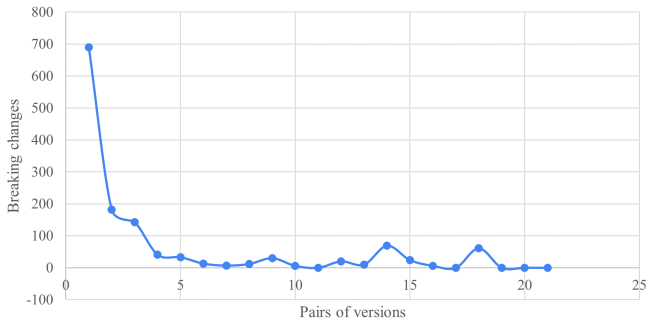

Figure 6. Scatter of pairs of versions and breaking changes

The scatter line shows a decreasing trend in most of the pairs of versions, this indicates that last versions have less BR changes than older ones. An assumption to explain this behaviour is the effort from the developers of this particular dataset to decrease the number of breaking changes over time by fixing the problems that affect the applications that consume the API, in order to guarantee the evolutionary improvement in each of the versions released. When performing the same analyses on other datasets outliers may be present since the decrease of breaking changes in new API versions depends on the API developers.

4.3.2. API level

Below is a chart that allows you to see the comparison between two specific APIs (azure.com:apimanagement-apimapis-network versus azure.com:apimanagement-apimsubscriptions-blue),

where both have two pairs of equal versions: *2018-06-01-preview-2019-01-01-01* and *2019-01-01-01-2019-12-01-01-preview*. Figure 7 shows the behavior of decreasing breaking changes in the first API (apimapis) and an increase of breaking changes in the second one (apimsubscriptions):

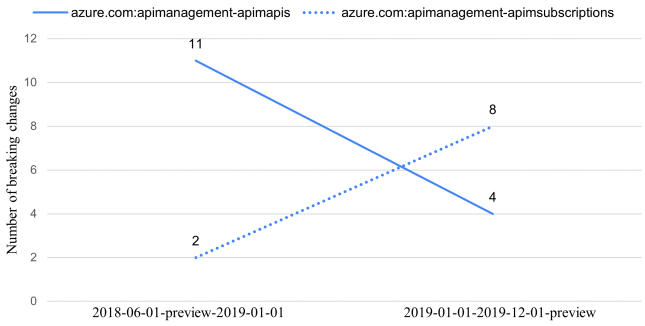

Figure 7. Comparison of breaking changes between two APIs.

Based on the three previous analyses, it is possible to obtain insights about the stability of an API. The first analysis helps one review the average number of endpoints changing in an API. The second analysis showed that for this dataset the breaking changes tend to decrease when a new version of an API is released. Finally, the last analysis shed light on the increase/decrease of the number of breaking changes in the different API version pairs.

5. Usefulness of the results

To validate whether the analysis presented so far was perceived as useful by practitioners, we prepared a survey to see how the previous charts could be insightful for a number of use case scenarios around API evolutions.

The survey was responded by ten practitioners (eight men and two women), all of them software engineers in the private sector and with 3-5 years working on the development and maintenance of APIs.

For each chart, we asked the participants two questions: i) what kind of decisions/processes would the charts help to make/perform; and ii) usefulness of the charts.

With respect to the first group of questions, we gave the participants four possible charts usage options and prompted them to suggest new usage ideas. The initial usages were: (U1) define work priorities to adjust the client software to the changes introduced in the API's latest version; (U2) define concrete actions to reduce the impact of changes found in client applications; (U3) identify the need to refactor the API to improve its maintainability; (U4) if the compared APIs satisfy the same functionality, the comparison could motivate to choose one API over the other because the first one changes less. The participants agreed with some of the usages listed before but also proposed the following two usages: (U5) identify the volatility/stability of an API over time; (U6) get an initial idea about the effort to adapt client applications in response to API changes. The more breaking changes, the greater the effort to move client software to the new version of the API. Table 3 shows for each chart the aforementioned usage scenarios selected by the

participants. On this table, an X on a row means that the particular chart is useful in that usage scenario.

Regarding the second question, the participants evaluated the usefulness of each chart by using a likert scale, with levels of agreement/disagreement being the dimension used. All the participants strongly agreed that the classification of changes in breaking and non-breaking changes (i.e., charts 3 and 4 in Table 3) are of great value for them. Participants perceived the usefulness/easiness of charts 6 and 7 in a lower amount compared to that of the other charts: 50% of the participants agreed, 30% remained neutral and the rest disagreed on the usability.

CHART	U1	U2	U3	U4	U5	U6
(1) Total changes per severity	X	X				X
(2) BR / NON-BR distribution	X	X	X			
(3) Single API stability	X	X	X			
(4) BR changes proportion	X		X	X	X	

Table 3. Chart Usage According to Practitioners

6. Threats to validity

Our work is subjected to both (1) internal validity (i.e., related to the inferences we made); and (2) external validity (i.e., generalization of our findings). Regarding the internal validity, it is worth mentioning that at the time we obtained the information from *APIsGuru*, the only APIs that had several version-pairs and that were specified in the OpenAPI version (i.e., v2.0), requested by the ADT tool, were those provided by Azure. This is a thread of validity but it was necessary at the time in order to obtain a set of APIs to analyze. Additionally, note that the APISGURU was selected because they provide publicly available APIs actively supported by developers. Also, the repository is continuously updated, and the API definitions are reviewed to ensure its correctness. The proposed taxonomy covers API changes presented by both other authors and us but additional changes may have been missed. As for the external validity, our study is based on a subset of OPENAPI APIs and therefore our results should not be generalized to any other kinds of APIs.

7. Related work

The topic of API evolution has been largely studied in the literature [Dig and Johnson 2006, Wu et al. 2016, Xavier et al. 2017, Sawant et al. 2019, D.Rivières 2019, Henkel and Diwan 2005, Hora et al. 2014, Nguyen et al. 2010, Zhong et al. 2009, Wu et al. 2014, Bartolomei et al. 2010, Lamothe et al. 2021]. Some of the earlier works focused on the evolution of software libraries and have served as inspiration for the taxonomy definition. More recent ones target specifically web-based APIs. We have identified two categories of works in this context: (i) those focusing on understanding the API evolution process by classifying API changes; and (ii) those that study API usages on client software by mining repositories. Our work falls in the intersection of these categories in the sense that it focuses on API evolution itself (excluding client applications) but the research method includes mining techniques. In what follows we compare our approach with the closely related literature.

Web API evolution. Regarding the works studying the API evolution process, [Li et al. 2013] establish Web API evolution patterns and how different those are

from the ones found on regular APIs. [Romano and Pinzger 2012] report the differences found between pairs of API versions represented in the Web Services Description Language (WSDL). The authors transform the representations into models and compare them using EMFCompare [Eclipse Fundation 2019]. [Koçi et al. 2019] present a classification that shows which REST API elements are prone to change. This classification takes into account paths, parameters, requests, and changes in authority level. [Wang et al. 2014] study online discussions about REST API changes by analyzing the StackOverflow questions. They identify 21 types of changes and claims that 7 of them are new compared with existing studies. [Polák and Holubová 2015] represent the initial version of a given API in a model called ReM. Whenever a new change requirement appears, the change is specified in the ReM model and generates a new version of said model, so that the users can trace the API evolution in time. The authors also provide model transformations that generate API source code, in a specified programming language, from a ReM model version.

In particular, our approach differs from existing related works in three key aspects. Firstly, we specialize in REST APIs, the most popular ones at the moment. The proposals of [Koçi et al. 2019] and [Wang et al. 2014] are the only ones that target this kind of APIs. Nevertheless, their change detection approach is entirely manual and based on the revision of documentation and API controllers used on client software, thus making their process technology specific. In contrast, our change identification process is automated and compares APIs using OPENAPI documents, which allow us to process APIs written in any programming language while focusing on structural changes. Second, we use a more extensive taxonomy of changes to detect and classify the API evolution. Third, our approach compares the APIs using a model-based representation of each API version. This allows us to further process the changes detected and add metadata to them. In our review, we found [Romano and Pinzger 2012] to be the only one that uses model-based comparisons to detect changes. However, they do not perform any additional processing on the comparison model returned by EMFCOMPARE. In contrast, we have added a step to the comparison process that enriches the EMFCOMPARE results with complex changes that go beyond the classical simple changes (i.e., delete, add, modify).

API usages in client software. [Wang et al. 2019] propose a mining engine called *Crmac* which recommends APIs based on regularity of use taking into account code from both open-source projects and local projects. [Sawant and Bacchelli 2015] analyze the use of certain APIs consumed by projects in GitHub. This approach differentiates from others in that it establishes the connection between the API class and the invocation in the client application. This allows developers to understand the use of an API with greater precision. [Lamothe 2020] establishes practical and scalable guidelines/tools regarding the evolution of APIs based on what is found mining public code repositories. The insights are delivered to both stakeholders: API developers and consumers. [Huppe et al. 2017] identify patterns of API usages, by utilizing genetic programming, to inform non-trivial patterns that allow developers who consume these APIs to understand their usage more efficiently. [Bae et al. 2014] detect possible misuses of Web APIs on client applications written in Javascript. [Yasmin et al. 2020] identify depreciated elements in different versions of REST APIs. The authors mined 1368 APIs from APIsGuru and determined that there are many problems related to the deprecation of REST APIs.

[Koçi et al. 2021b, Koçi et al. 2021a][Koçi et al. 2020] propose to inform providers about API usage patterns and usability by regarding the interaction of consumers with APIs reported in logs. Finally, [Di Lauro et al. 2021] study evolution by mining open-source API versions specified in the OpenAPI standard. Their analyzes focuses on the following three aspects: overall number of changes, API age, and API growth.

Like most aforementioned approaches, we mine public API datasets. However, we focus on the API versions themselves by plotting the mining results in graphics, at aggregated (repository) and isolated level (API). Our approach differentiates from others in that it classifies changes in a taxonomy that has categories (either breaking or non breaking) and subcategories (e.g., rename parameter, delete path, etc.). This information may help developers and consumers to understand API evolution and make decision to improve maintainability as shown in the validation. The analyses of API usages in client applications is out of the paper scope.

8. Conclusion and future work

We mined a set of APIs from APIsGuru to better understand the common changes when evolving APIs. We concluded that the category of breaking changes has more types of changes, thus showing that the adaptation of client applications to API evolution is a critical step in any migration/evolution process and should be planned as such. Further validation with practitioners showed that the results and proposed charts are useful to assist them in their API adaptation process, mainly to help to prioritize the required adaptation work. In addition, we concluded for the API Evolution Analysis section that most of the APIs in the dataset were affected by some change and that most of them were non-breaking changes. We also observed that the most repeated breaking change was *Increase Number of Parameters* with 658 occurrences and the most repeated non-breaking change was *Rename Parameter* with 1473 occurrences. Lastly, we verified that the APIs of the dataset under study follow the behavior of decreasing the number of breaking changes in each new version that is released. This is a good indicator for the dataset, most API releases tended to decrease their breaking changes, which is an ideal behavior and benefits those who consume these APIs. However, this behavior is not always the case, therefore, good practices should be followed when releasing new versions of an API.

Regarding the further work, we would like to develop new methods and tools to recommend/apply client software adaptations to the frequent API changes found in this analysis. Another possible step would be to prioritize the adaptations based on breaking changes severity. We are interested in supporting other API definition formats (e.g., OpenAPI v3.0). Also, we would like to explore the change detection beyond static analysis of the APIs definition. For example, as depicted in our taxonomy section, a change in the authentication model may only be detectable at runtime by perusing the API HTTP responses.

References

Bae, S., Cho, H., Lim, I., and Ryu, S. (2014). Safewapi: Web api misuse detector for web applications. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 16-21-November-2014:507–517.

Bartolomei, T. T., Czarnecki, K., and Lämmel, R. (2010). Swing to SWT and back: Patterns for API migration by wrapping. In *26th IEEE International Conference on*

Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania, pages 1–10.

Cao, H., Falleri, J.-R., and Blanc, X. (2017). Automated generation of rest api specification from plain html documentation. In *Service-Oriented Computing*, pages 453–461, Cham. Springer International Publishing.

Di Lauro, F., Serbout, S., and Pautasso, C. (2021). Towards large-scale empirical assessment of web apis evolution. In Brambilla, M., Chbeir, R., Frasincar, F., and Manolescu, I., editors, *Web Engineering*, pages 124–138, Cham. Springer International Publishing.

Dig, D. and Johnson, R. (2006). How do apis evolve? a story of refactoring. *J. Softw. Maint. Evol.*, 18.

D.Rivières (2019). Evolvingjava-basedapis2.

Eclipse Fundation (2019). Emf compare.

Halili, F. and Ramadani, E. (2018). Web services: A comparison of soap and rest services. *Modern Applied Science*, 12.

Henkel, J. and Diwan, A. (2005). Catchup! capturing and replaying refactorings to support api evolution. *Int. Conf. on Software Engineering*, 27:274 – 283.

Hora, A., Etien, A., Anquetil, N., Ducasse, S., and Valente, M. (2014). Apievolutionminer: Keeping api evolution under control. *Int. Conf. on Software Maintenance, Reengineering, and Reverse Engineering*.

Huppe, S., Saied, M. A., and Sahraoui, H. (2017). Mining complex temporal api usage patterns: An evolutionary approach. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 274–276.

Koçi, R., Franch, X., Jovanovic, P., and Abelló, A. (2021a). Improving web api usage logging. In Cherfi, S., Perini, A., and Nurcan, S., editors, *Research Challenges in Information Science*, pages 623–629, Cham. Springer International Publishing.

Koçi, R., Franch, X., Jovanovic, P., and Abelló, A. (2021b). Patternlens: Inferring evolutive patterns from web api usage logs. In Nurcan, S. and Korthaus, A., editors, *Intelligent Information Systems*, pages 146–153, Cham. Springer International Publishing.

Koçi, R., Franch, X., Jovanovic, P., and Abelló, A. (2019). Classification of changes in api evolution. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 243–249.

Koçi, R., Franch, X., Jovanovic, P., and Abelló, A. (2020). A data-driven approach to measure the usability of web apis. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 64–71.

Lamothe, M. (2020). Bridging the divide between api users and api developers by mining public code repositories. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 178–181.

Lamothe, M., Guéhéneuc, Y.-G., and Shang, W. (2021). A systematic review of api evolution literature. *ACM Comput. Surv.*, 54(8).

Li, J., Xiong, Y., Liu, X., and Zhang, L. (2013). How does web service api evolution affect clients? *Int. Conf. on Web Services*.

Nguyen, H., Nguyen, T. T., Jr., G. W., Nguyen, A., Kim, M., and Nguyen, T. (2010). A graph-based approach to api usage adaptation. *ACM SIGPLAN Notices - OOPSLA '10*, 45:302–321.

Polák, M. and Holubová, I. (2015). Rest api management and evolution using mda. *Int. Conf. Proceeding Series*, 13:102–109.

Romano, D. and Pinzger, M. (2012). Analyzing the evolution of web services using fine-grained changes. In *2012 IEEE 19th International Conference on Web Services*, pages 392–399.

Sawant, A., Robbes, R., and Bacchelli, A. (2019). To react, or not to react: Patterns of reaction to api deprecation. *Empiric Software Engineering*, 24:3824–3870.

Sawant, A. A. and Bacchelli, A. (2015). A dataset for api usage. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 506–509.

Sohan, S., Anslow, C., and Maurer, F. (2015). A case study of web api evolution. *IEEE World Congress on Services*.

SOM RESEARCH LAB (2018). Apicomposer.

Wang, C., Yang, Y., Liu, H., and Kang, L. (2019). Statistical api completion based on code relevance mining. In *2019 IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT)*, pages 7–13.

Wang, S., Keivanloo, I., and Zou, Y. (2014). How do developers react to restful api evolution? *Int. Conf. in Service-Oriented Computing*, 8831:245–259.

Wu, W., Adams, B., Guéhéneuc, Y., and Antoniol, G. (2014). Acua: Api change and usage auditor. In *Int. Conf. on Source Code Analysis and Manipulation*, pages 89–94.

Wu, W., Khomh, F., Adams, B., Guéhéneuc, Y., and Antoniol, G. (2016). An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Emp. Softw. Eng.*, 21:2366–2412.

Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017). Historical and impact analysis of api breaking changes: A large-scale study. *Int. Conf. on Software Analysis, Evolution and Reengineering*, pages 138–147.

Yasmin, J., Tian, Y., and Yang, J. (2020). A first look at the deprecation of restful apis: An empirical study. In *2020 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME 2020)*, Proceedings-IEEE International Conference on Software Maintenance, pages 151–161. IEEE; IEEE Comp Soc. 36th IEEE International Conference on Software Maintenance and Evolution (ICSME), ELECTR NETWORK, SEP 27-OCT 03, 2020.

Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). Mapo: Mining and recommending api usage patterns. *Europ. Conf. on Object-Oriented Programming*, 5653:318–343.