Classification of Changes in API Evolution

Rediana Koçi, Xavier Franch, Petar Jovanovic, Alberto Abelló
Universitat Politècnica de Catalunya, BarcelonaTech
{koci, franch, petar, aabello}@essi.upc.edu

Abstract—**Applications typically communicate with each other, accessing and exposing data and features by using Application Programming Interfaces (APIs). Even though API consumers expect APIs to be steady and well established, APIs are prone to continuous changes, experiencing different evolutive phases through their lifecycle. These changes are of different types, caused by different needs and are affecting consumers in different ways. In this paper, we identify and classify the changes that often happen to APIs, and investigate how all these changes are reflected in the documentation, release notes, issue tracker and API usage logs. The analysis of each step of a change, from its implementation to the impact that it has on API consumers, will help us to have a bigger picture of API evolution. Thus, we review the current state of the art in API evolution and, as a result, we define a classification framework considering both the changes that may occur to APIs and the reasons behind them. In addition, we exemplify the framework using a software platform offering a Web API, called District Health Information System (DHIS2), used collaboratively by several departments of World Health Organization (WHO).**

Index Terms—**API evolution, API changes classification, log mining, issue tracker.**

I. Introduction

Nowadays, Application Programming Interfaces (APIs) are being broadly used [16]. The main reason for this success is that APIs provide advantages to their consumers (software developers) and their producers (companies and institutions that expose their organizational data). Software developers that use APIs do not have to start from scratch when coding their applications. By outsourcing some functionality to the API, they can speed up their work focusing on other requirements. On the other hand, by making available their API, organizations can increase the customer reach of their brand or can create a new revenue stream by monetizing the API.

In an ideal world, the cooperation between API producers and API consumers could be described as follows: API producers develop a stable API, providing very detailed and helpful documentation, so API consumers use it without difficulties, while further improvements of the API do not affect them. In practice, the opposite usually happens: APIs are prone to continuous changes, often backwards incompatible and supported by poor documentation. All of this has a negative impact on API consumers [1], [3], [7].

Our main objective is to give an overall view of how an API change is reflected not only in its implementation but in four artifacts, namely release notes, API documentation, issue tracker, and versioning system. We firstly identify and classify the changes that often happen to APIs, by analyzing the API controller. The syntax of API, that consumers use in the calls, is implemented in the controller code. It handles the request/response to and from API, so every change made on it will impact the consumers. Then, we analyze the artifacts, to see where and how API producers explicitly introduce them. We refer to API documentation and release notes as two main sources of information for API consumers when they integrate with an API or upgrade to a new version of it [6], [8]. On the other hand, we take in study issue tracker and versioning system as two important tools used by API producers while developing and evolving APIs [11], [18]. We evaluate the impact these changes have on consumers by analyzing the API log files, as they contain the calls that API consumers make to the API. This way we avoid analyzing the code of API consumers' applications, which often is not available.

We review the current state of the art in API evolution and, as a result, we define a classification framework considering changes that may occur to APIs, the causes behind them, and the impact they have on API consumers. In addition, we apply our framework on a real world use case. Analyzing the complete API evolution lifecycle, from raising the issue, through its implementation, documentation, publishing it in the release notes, and finally analyzing its usage though API calls, helped us to better understand the impact that this process has on the API consumers. Throughout this paper we focus on the Web APIs (APIs over the Internet), and for the sake of simplicity we refer to them as API. We introduce and further use the following concepts:

- API producers - those who develop and expose the API.
- API consumers - those who develop applications that rely on and consume the API.
- API change - a change in API declaration level.
- API controller - handlers of incoming/outgoing HTTP request/response.
- API artifacts - sources where API producers explicitly introduce information about evolution, like release notes, documentation, issue tracker and versioning systems.

Our study is driven by the following research questions:

- RQ1: Which are the changes that happen to APIs when they evolve?
- RQ2: How are the changes that happen to APIs reflected in different API artifacts?
- RQ3: Which are the causes of the API changes?
- RQ4: To what extent are the API changes reflected in the usage logs?

In summary, this paper makes the following contributions:

- Identifies changes that can happen to APIs.

- Classifies the API changes depending on their causes.
- Analyzes how these changes are reflected in documentation, release notes, issue tracker and log files.

Outline. In section II we give an overview of the state of the art of the API evolution. In section III, we present the classifications derived from a literature review. In section IV, we describe our methodology and apply it on a real world use case. Section V discusses our findings, while section VI concludes the paper and discusses future work.

II. Related Work

We analyze the related work mainly focusing on two research lines, API evolution and API usage.

A. API evolution

Evolution of APIs has gained considerable attention from researchers recently covering different aspects of this cumbersome process [3], [7], [10], [14], [17]. Studies have been conducted to identify the changes that occur to APIs from older to newer versions [3], [10]. Wang et al. [10] gave a catalogue of changes that happen to APIs and consumers reactions but did not provide any suggestions on why they happen or how to deal with them. Dig and Johnson [3] manually identified the changes of five Java APIs. Based on the impact these changes have on API consumers, they classified them as Non-Breaking API Changes and Breaking API Changes. More than 80% of the breaking changes were refactorings, thus they suggested refactoring-based migration tools for applications update. Li et al. [7] made similar recommendations. After analyzing the changes in five web APIs, they gave suggestions for designing migration tools to better help the migration of clients.

A lot of effort is put in analyzing the impact that API changes have on consumers applications. Robbes et al. [20] assessed the impact of API deprecation on a Smalltalk ecosystems in terms of frequency, magnitude, duration, adaptation, and consistency of the ripple effect (adaption to API deprecation). Espinha et al. [1], [2] in their exploratory studies interviewed API consumers to share their struggles and experiences during API evolution. They measured the impact of evolution on the client side by analyzing how much code had been changed. The study showed that the lack of an industry standard and high frequency of changes has led to the decrease of satisfaction of Web APIs consumers. Thus, they recommended not to change the APIs too often and to perform blackout tests (shutdown of older version of API in a short time frame).

Most of the above works pay attention to the API consumer' side, leaving aside API producer' side. However, the latter also face challenges and difficulties in managing and evolving API. Xavier et al. [5] made a survey to reveal the reasons why producers break APIs. Brito et al. [4] did a reason-based classification of changes in APIs of 400 libraries. In both of above-mentioned studies, they asked directly the API developers, about their motivation to change the API. The reasons given were the need to: implement new features, fix bugs, simplify the API, improve maintainability and to refactor (to improve the internal code). For each motivation, they gave the changes that API producers performed in order to achieve the desired results, but only Java libraries were analyzed in both of these two surveys. Web APIs compared to library APIs, present different challenges, not only for consumers but also for their producers (e.g., API traffic).

Contrary to our work, that gives an overall view of all aspects of API when changes are performed, the mentioned research works are focused on a specific aspect of API. They do manual monitoring of the changes or interviews with API producers and consumers, to give a summary of changes that happen to APIs, and sets of good practices to make migration less painful. However, most of these approaches do not consider the way APIs are consumed. Indeed, the information revealed from the usage of APIs will help us to understand the impact that changes have on their consumers.

B. API usage

Different studies exist in analyzing API usage, i.e. the ways consumers use the API. Ed-douibi et al. [12] analyzed the API calls to present an example-driven discovery process that generates model-based OpenAPI specifications for REST Web APIs. With their findings, they aimed to help developers in speeding up the process of interacting with the API. Zhong et al. [13] developed an API usage mining framework and a tool called MAPO for mining API usage patterns automatically from code snippets. Their goal was to help programmers understand API usages and write API client code more effectively. Wu et al. [14] analyzed and classified API changes and usages together, but they did not focus on the API call, but on the client programs. They provided suggestions for developers and researchers to reduce the impact of API evolution through language mechanisms and design strategies.

From our point of view, analyzing consumers' applications code can be an unrealistic approach. Considering that the code is not always available, using it as input is not always possible. Moreover, the detected patterns cannot be generalized for all the consumers applications of the APIs in study. Each of them might have their own way to use the API, in the form of different sequences of calls. Thus, we aim to investigate the impact of the evolutive actions by analyzing consumers behaviour from the API usage logs files. These files contain the calls of API consumers to the API. Combining knowledge on API evolution and its impact in the API usage will be beneficial in understanding the overall API evolution process.

III. Classification of changes

During their lifecycle, APIs experience several evolutive iterations. Throughout these phases, API producers perform different changes, which sometimes make the release of new versions of the API necessary. According to Semantic Version[1] scheme, which uses a sequence of three digits (Major.Minor.Patch) to control the versions, when producers perform backward compatible bug fixes, they launch a

[1]https://semver.org

new Patch version, when they perform backward compatible changes they launch a new Minor version, and when they perform non backward compatible changes, they launch a new Major version. The two first kind of changes, for Patch and Minor versions, are non-breaking changes. They will not prevent existing API consumers' applications from functioning after the upgrade, while API consumers can optionally learn and modify their code to benefit from new features and improvements. Conversely, the last type of changes, introduced in Major versions, are breaking changes. After upgrading to Major versions, API consumers are required to modify their code to comply with the new API version. It is important and effective to look and analyze the history of these changes in order to assist and anticipate further evolution of APIs.

A. Which are the changes that happen to APIs?

By observing the evolution process from points of view of both API producers or consumers, we can apply different classifications to these changes.

From the consumer's point of view, referring to the compatibility of the new version with the previous ones, API changes can be divided in two types: breaking changes and not breaking changes [3]. Next, breaking changes can be classified in changes that affect the behaviour of the API (pre and post conditions, changes in API response) or the syntax of API.

From the producer point of view, changes can be classified based on the causes of these changes (e.g., to add new features, to simplify the API), the changed API elements (e.g., changes on attributes, on methods or on classes) or the actions performed on them (e.g., moving elements, adding new ones, or refactoring). These classifications point out which part of APIs are more stable and which ones are more change prone during their lifecycle. Brito et al. [4] used two first classifications in their work. They referred to changes of different API elements (types, methods and fields) and causes of the changes. Sohan et al. [17] classified the change patterns based on the action performed on the API element (Add[APIElements], Remove[APIElements], Change[APIElements] etc.).

In this paper, we focus on the changes that affect the syntax of the API (i.e., declaration level). Consumers interact with a system components using their API, i.e., the interface of the component. Thus, they are directly impacted by changes that affect their syntax. To detect these kind of changes, we suggest the comparison of API controller of two consecutive versions. The only drawback of this option is that the API controller is available only for open source projects. We look for changes that can be performed on API elements as follows:

- *API endpoints* are URLs to access API resources (data, functionality). API producers expose resources by providing endpoints to access them. They can disconnect endpoints (remove), add new ones, rename, or even replace an existing endpoint with a new one.
- *Parameters* are used to refine resources. They can be path parameters (part of the request body separated by "/") or query parameters (part of request body after '?' in a key=value form). Parameters can be optional or mandatory. New parameters can be added to resources, existing ones can be removed, renamed, change type or even change from optional to required, and vice versa.
- *Parameters value*. We can pass value from a predefined set of values to some parameters (e.g., parameter timePeriod can have value from {week, month, year}). If not defined in the call, then the default value is passed to them. Both the set of possible values and the default one can change.
- *Request methods* represent the action we make to the resource, like GET, POST, DELETE, UPDATE, etc. While evolving their APIs, producers can support or unsupport methods for a specific resource.
- Changes in *authority levels*. To interact with specific resources, users should have the required authority level. The set of authority levels for a resource can change by adding new ones, removing or just change the needed authority.

This classification guides us in tracking the changes, helping in channelling our investigations for each API element.

B. How are the changes that happen to APIs reflected in different API artifacts?

While performing changes on the API, API producers make use of different tools and have different practices in publishing, tracking, storing and communicating these changes to consumers. We refer in overall to four different API artifacts, namely release notes, API documentation, issue tracker, and versioning systems, to see how evolution is reflected on them and to what extent they document the changes.

- **Release notes** are documents that accompany the release of new software versions to communicate to their consumers the new changes. Abebe et al. [8] made an empirical study to analyze the content and structure of different release notes and noted that most of them contained only a limited number of changes (from 6% to 26% of all the new issues). They listed different factors on which release notes' writers base their decision to select the issues, like: issue type, issue priority, number of modified files, number of comments in issue tracker, size of issue description, number of days to address the issue, experience of issue reporter, etc [8].
- **API Documentation** has a technical nature. They provide detailed information about API elements, such as endpoints, resources, fields, types, and parameters, often showing examples [6]. API documentation is usually manually written, and sometimes this process is not synchronized with new version release. All these result in outdated documentation [15]. Actually, obsolescence in documentation is one of the most important concerns of API consumers when they upgrade to new API versions [1], [2], [19]. Uddin and Robillard [19] suggest not to expect all the changes to be present in the documentation. They pointed out the importance of changes that break the backward compatibility to be especially documented.

- **Issue tracker** systems are tools that help teams and organizations record, keep track and manage issues like bugs, features and requests. Bertram et al. [18] in their study conducted interviews with developers who used an issue tracker. They considered the comments section as one of the most valuable parts of issue trackers. These comments, made in form of discussions between developers and everyone interested in that issue, are plenty of valuable information. When the reason of the issue opened is not specified in the description, referring to comments can give a better understanding of it. However, issue tracker systems are not always accessible. They are open only in case of open source APIs.
- **Versioning systems'** history logs contain information for every change in the repository. Developers can associate comments to their commits to explain their actions. These comments, depending on developers style of coding and project regulation, can vary from simple and short descriptions, to more detailed ones.

C. Which are the causes of the API changes?

When performing evolutive actions, API producers are driven by different reasons, e.g., to add new features, to fix bugs, to simplify the API, to improve maintainability, or to improve the security of the API [3]–[5]. Their identification completes the big picture of the evolution of API.

Changes and causes have a many-to-many relationship. When API producers apply changes in their APIs, even though driven by different reasons, the set of changes can be the same. This is even more true in bug fixing, because bugs can have different nature (e.g., logical errors, compilation errors, functional errors, or calculation problems). Changes done in order to fix a compilation error can be different from those done in errors caused by faulty calculations. Moreover, classifying all the changes in new features, bug fixes, simplifications and maintainability improvements can result in a too coarse grained classification. Additionally, every new feature is related with a specific component of API, so this classification can go finer. In our work, we classify the changes based on API's aspects they affect. This will permit us to observe also trends in API changes: which aspects of APIs are more prone to change during API lifecycle. We adopted the usability taxonomy developed by Mosqueira-Rey et al. [9] to classify changes based on the target usability aspect of API they aim to change, as follows:

- Know-ability - changes that aim to improve the ability of API to be easily understood and learned by consumers.
- Operability - changes that aim to enrich the API with new features and functionalities, fulfilling the needs of different users.
- Efficiency - changes that aim to improve the performance of the API and its consumers in terms of effort and time spent in interacting with the API resources.
- Robustness - changes that aim to increase the capacity of the API to prevent errors from its consumers or third parties.
- Safety - changes that aim to increase the safety, security, privacy and confidentiality of API resources and API consumers.
- Subjective satisfaction - changes that aim to improve the aesthetic of system and increase the interest of consumers in using it.

D. How are the API changes reflected in the usage logs?

API consumers access APIs via HTTP requests, in the form of a URL. These access logs can be obtained by monitoring the API traffic in either the server side (provider) or the consumers side. A log file contains log entries, each of which represents a call to an API endpoint, Fig. 1.

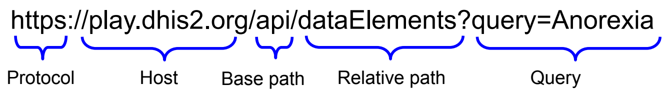

Fig. 1. An URL to call an API.

We can refer to the access logs as traces that consumers leave after using the API. If the information in these traces is analyzed in the proper way, it can reveal useful knowledge. They show which API endpoints the consumer has accessed, in which order, and with which parameters to filter the response.

Almost every part of the API call can be prone to change when the API evolves. Some API providers choose to specify the version of the API in the URL, as part of the base path. Thus, when consumers have to upgrade to a new release, they should change the URL of every call they have made to the API in their applications. The relative paths are also prone to change. In the API call, the relative path is the API resource the consumers want to access. The query part of the call, in the form of key-value, contains parameters of the resource. These parameters can be optional or mandatory.

IV. Use case: DHIS2 API

A. Methodology - Exploring evolved APIs

In our work, we study APIs evolution and aim to build our concepts on changes that happen to them by exploring different aspects in existing literature and the data from our use case.

First of all we identify the changes by comparing two consecutive versions of API code. Then we analyze different API artifacts, to see in what extent are the changes documented in them. We do this step manually, and for the sake of completeness we refer to different sources. We classify the changes based on their causes and, at the same time, we analyze the API usage logs to see the impact that these API changes have on consumers. At the end, to better conceptualize the cause-effect relationship of the changes in the evolution process, we combine the two classifications, the type of changes and their causes to find correlations between them and the effect they have on API consumers (detected in the logs). This conjunction emphasize the impact each class of changes has on the consumers side, providing us clues on changes that can be identified from the logs.

B. DHIS2 use case.

We applied our approach on the API of DHIS2[2], which is an open source, web-based health management information system, used by more than 60 native applications. It has a strong and open API, built under REST architectural style. We took on study version 2.27 of the API, released on 01.06.2017 and analyzed the usage logs from 11.06.2016 to 29.11.2018 in WHO installation.

1) Which are the changes that happen to APIs?: In order to get the whole set of changes we compared API controller of version 2.26 and 2.27. It handles the incoming HTTP requests and sends responses back to the caller. As we were interested in changes that affect the API syntax, this level of comparison provided the desired set of changes.

TABLE I
DHIS2 2.27 CHANGES FROM CONTROLLER COMPARISON.

Type of change	**Occurrence**
New parameter	19
New endpoint	10
Remove Endpoint	5
New authority	2
Change authority	1
Support Request Method	1
Total	**38**

In overall, we found 38 changes introduced in the new API release, belonging to six different types of changes (Table I).

- There were 19 new parameters that were added to the existing endpoints. These parameters provided pagination, ordering and filtering of the information returned by endpoints.
- We found 10 new endpoints in the new version. They provided new features to the API, like the possibility to send notifications (`sendNotifications`), to validate the new password (`validatePassword`) etc.
- 5 endpoints were deleted from 2.27 version. API producers claimed to have removed not used endpoints or endpoints whose functionality were already replaced in the previous versions.
- 2 new authority were added in order to access two existing endpoints.
- The authority needed to POST to predictor/run endpoint changed from `'F_PREDICTOR_ADD'` to `'F_PREDICTOR_RUN'`.
- For `systemID` endpoint, POST method was supported in the new version.

2) How are the changes that happen to APIs reflected in different API artifacts?:

- **Release notes**: We analyzed the release notes for the version 2.27 of DHIS2 system. They were organized in different sections, each of which covered changes and updates for different aspects of the systems, e.g., Analytic Features, Tracker Features, General Features, Server Admin Features and Web API Features. Under each item, they provided additional information in the form of demo examples, screenshots, or links to documentation.
We analyzed the Web API Features section of the release notes[3]. Every item was presented in the form of a title and a short description, for example:
"*Min-max data element values:* A new endpoint for setting and retrieving min-max data element values is introduced at /api/minMaxDataElements."
The description lacks in explanation on how to use the new endpoint, its attributes, etc. Beside this, only six changes were introduced in the release notes.
- **API Documentation**: We compared the documentation of version 2.26 and 2.27 and noted that sometimes they were not updated. We found features of older versions to appear for the first time in the latest documentation. For example, `verifyPassword` is an endpoint that is used to verify the old password when the user wants to renew it. Even though this was live in version 2.25[4], it was documented for the first time in 2.27 documentation. There were also changes not yet documented like the removal of `ProgramStageDataElements` endpoint. It has completely disappeared in version 2.27, and this was documented nowhere[5]. From the other hand, we saw that changes that appeared in documentation were explained in details. For example, for the new endpoint `deletedObjects`, a whole paragraph were added in the documentation, describing how it works, and giving example of calls to the new endpoint.
- **Issue tracker**: DHIS2 uses JIRA as issue tracker system. We extracted the issues of 2.27 release and filtered them based on issue type (Bug, Feature, Design, Epic, Test, User Story), fixVersion (all releases), and Component (Application components, API components, Test, Documentation, Frontend, Backend). We queried the issues with the following constraints: Issue type = Bugs or Feature, fixVersion = 2.27, Component like API* (issues related with changes in API component level).
Even though the extracted information was more detailed, it was not too accurate since issues are manually opened by developers. Sometimes, they were not linked to the right version (`fixVersion` may be null) or even though the issue was related to API, the Component field was not filled properly, thus resulting in an incomplete list.
- **Versioning systems**. We referred to Github commits' history, to find information about causes of changes in the API. Commits at versioning systems often did not have comments or they were too short. For some of the commits related to bugs fixes, in the comments we could find the issue ID of the issue opened at JIRA.

3) Which are the causes of the API changes?: From the four API artifacts in study we referred to issue tracker to find the causes of the changes. Since only a small number of changes appeared in release notes and API documentation, and commits at versioning systems lacked in comments, the

[2]https://www.dhis2.org/about

[3]https://www.dhis2.org/downloads

[4]https://github.com/dhis2

[5]https://jira.dhis2.org/browse/DHIS2-1939

only possibility left was issue tracker.

The reasons were explained in issues description or in the comments discussion. We checked every issue manually. As all project's contributors (with or without technical background) can open issues at JIRA, the language used by them was not standard, thus making difficult to create a unified set of causes. If we would rely on JIRA issue types, we would have a very coarse grained classification: bug fixes and features. We used the usability taxonomy of Mosqueira-Rey et al. [9], and were able to fit every change in these classification, as in Table II.

TABLE II
DHIS2 2.27 JIRA FEATURE CLASSIFICATION.

API improved aspect	2.27 Release
Operability	26
Robustness	11
Efficiency	9
Knowability	6
Safety	8
Subjective Satisfaction	3
Total	**63**

The changes extracted from JIRA are more than those identified from controller comparison because in JIRA issues are related to changes not only in the syntax of API.

4) How are the API changes reflected in the usage logs?: We analyzed the Apache server logs of the DHIS2 system. The log entries contained information about client IP address, request time, the request, the status code that the server sends back to the client, and the size of the object returned.

Here is an example of how an API call looks:

```
http://.../api/dataElements.json?query=Anorexia
```

Its corresponding entry in the log files:

```
147.83.72.200 [19/Mar/2019:10:21:22 +0100]
"GET /api/dataElements.json?query=Anorexia
HTTP/1.1" 200 175
```

We checked how the already done changes gathered from the first two steps, were reflected in the API calls. From 38 changes extracted from the API controller, only 10 of them where adapted from the API consumers, so we found traces of only 10 changes in the logs. Actually all these changes should appear in the logs, but as DHIS2 support four last versions of the API, its costumers delay the upgrade process. From these 10 changes (3 new endpoints and 7 new parameters), six of them were documented in at least one of the artifacts. Nevertheless we cannot extrapolate, from these few data about the type of change or being it documented or not, the fact that consumers use it or not.

In 2.27 release notes[6] a new feature was introduced:

"Min-max data element values: A new endpoint for setting and retrieving min-max data element values is introduced at /api/minMaxDataElements."

It appeared in the release notes and also in the documentation of version 2.27. For our surprise, all the calls made to the new endpoint `minMaxDataElements` got a 404 status code: client side error. This can be an indicator that consumers do not refer to documentation or that the documentation is not enough clear and lacks in examples. We saw that this was common with the new endpoints (e.g., also with `ProgramIndicatorGroup`, a new endpoint introduced in 2.27 version): it took time for consumers to properly use them.

In 2.27 version, a new query parameter was added to analytics endpoint, `hideEmptyColumns`. This parameter, when true, excludes from the response the columns with only null values. If we look at the calls to the analytics endpoint, without and with this parameter specified, we can see that the object size returned in the second case is significantly reduced. The same effect had the use of new parameters that provided pagination (`paging`), optimizing the API response.

[6]https://www.dhis2.org/downloads

V. Discussion

Within this work we presented an overview of API evolution. We performed manual analysis of API controller and four API artifacts, to identify and classify changes that happen to APIs and to investigate their impact on API consumers.

RQ1. We considered as ground truth the changes extracted from API controller comparison. Accessing it is only possible for open source APIs, but considering the incompleteness of the other artifacts, it is the only one that can provide the whole set of changes. Half of the changes introduced in the new release were new parameters. Parameters are usually used to filter the information of API resources. Thus, these additions can be explained as a way to provide new functionalities without splitting and rearranging the existing endpoints, avoiding this way breaking changes. The second most present change was the addition of new endpoints. In overall, more than 75% of changes were non breaking ones. This can explain somehow the fact that most API consumers had not been upgraded with the latest version of the API. API consumers delay the upgrade process until they need the new features, the fixed bugs, or until the API producers stop supporting the old versions of API.

RQ2. After analyzing different artifacts of API, we saw that less than 50% of the changes were documented in them. From 38 changes extracted from the controller, only 17 were documented in at least one of the artifacts (release notes, documentation, issue tracker). Approximately 5% of the changes (2 out of 38) were reflected on all the artifacts. Both of them were new endpoints. Even thought it is highly recommended for breaking changes to be documented [1], [2], [17], the addition of new elements is also seen from API producers as important to be documented. This way, API consumers can learn about these new elements and how to use them. We choose to take in study release notes, API documentation, issue tracker, and versioning system, as a representative set of sources from which we would be able to observe the API evolution. Nevertheless, we do not exclude the existence of other artifacts, like changelogs, migration guides, mailing lists of consumers of API, API official web-page, etc. As mentioned before, how these artifacts are maintained or used depends on the project conventions. For DHIS2 project, these four selected were the most complete and used ones.

RQ3. In previous studies [4], [5], interviews were conducted to API producers, in order to get their intention for every change. Even though this is the most straightforward way, this information can be found also in API artifacts, where API providers provide hints about the causes of the changes they perform. They tend to give such information in the issue tracker system (i.e., issue description, comments section). Their interactive and collaborative nature, mostly on comments section, makes these tools useful not only for API producers while evolving the API, but also for API consumers, giving them the possibility to get more clues about the changes.

RQ4. We analyzed the impact of the changes on consumers side, by looking at the API usage logs. We extracted the API log files from the server and examined them based on the changes extracted previously from the controller. Even though we had the set of changes, we were not able to see how all of them were reflected in the API calls. DHIS2 supports four last versions of the API, and few consumers did the upgrade to the new releases, thus benefiting from the new introduced features. Until API providers support the previous versions, consumers will not feel urge to upgrade, unless they really need the new features or the critical bugs fixed.

Threats to validity. In terms of validity, the main threat to external validity is that we take in study one API. Moreover, we focus only on syntactical changes of API. Future work need to be carried out to increase the data set in order to obtain more generalizable results. The main threat to internal validity is relate to the amount of manual work done. To alleviate this threat, we analyzed each change in different API artifacts.

VI. Conclusion and future work

After reviewing the state of the art, we applied a use case on API evolution. We investigated how this process is documented and reflected in different API artifacts, and highlighted some problematic aspects in them. We identified the changes that are usually performed on API, and classified them based on their types and causes. We investigated how these changes are reflected in the API calls, and how the type of change and its cause can affect the consumers.

In our future work, we will expand the analysis, by adding other use cases, to obtain more generalizable results. We will focus not only on changes on API syntax, but also changes on API behaviour. By getting the insights about how the changes are reflected in API usage logs, our next goal is to further scrutinize the logs in order to find the patterns and anticipate evolutive changes. Our work in analyzing the API evolution and how it is handled from both sides, producers and consumers, is a necessary step in understanding and further automating the API evolution process, which is essential for efficient and consistent API provisioning.

Acknowledgment

We thank the Neglected Tropical Diseases (NTD) department at WHO, for providing the use case. This work is supported by GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación under project TIN2016-79269-R.

References

[1] Espinha, T., Zaidman, A., Gross, H.G. “Web API growing pains: Stories from client developers and their code.” In Proceedings of IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pp. 84-93, 2014.

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

[3] Dig, D., Johnson, R. E. “How do APIs evolve? A story of refactoring.” In Journal of Software Maintenance, vol. 18, pp. 83-107, 2006.

[4] Brito, A., Xavier, L., Hora, A.C., Valente, M.T. “Why and how Java developers break APIs.” In Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 255-265, 2018.

[5] Xavier, L., Hora, A., Valente, M. T. “Why do we break APIs? First answers from developers.” In Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 392-396, 2017.

[6] Vasudesan, K. “What is API Documentation, and Why It Matters?” Swager, 06-06-2017. Available: https://swagger.io/blog/api-documentation/what-is-api-documentation-and-why-it-matters. [Accessed: 23-05-2019]

[7] Li, J., Xiong, Y., Liu, X., Zhang, L. “How does web service API evolution affect clients?” In Proceedings of International Conference on Web Services (ICWS), pp. 300-307, 2013.

[8] Abebe, S.L., Ali, N., Hassan, A.E. “An empirical study of software release notes.” In Journal of Empirical Software Engineering, vol. 21, pp. 1107-1142, 2015.

[9] Mosqueira-Rey, E., Alonso-Ríos, D., Moret-Bonillo, V., Fernández-Varela, V., Álvarez-Estévez, D. “A systematic approach to API usability: Taxonomy-derived criteria and a case study.” In Journal of Information and Software Technology, vol. 97, pp. 46 - 63, 2018.

[10] Wang S., Keivanloo I., Zou Y. “How Do Developers React to RESTful API Evolution?” In Proceedings of International Conference on Service-Oriented Computing (ICSOC), pp. 245-259, 2014.

[11] Hattori, L. P., Lanza, M. “On the nature of commits.” In Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 63-71, 2008.

[12] Ed-douibi H., Cnovas Izquierdo J.L., Cabot J. “Example-Driven Web API Specification Discovery.” In Proceedings of European Conference on Modelling Foundations and Applications (ECMFA), pp. 267-284, 2017.

[13] Zhong H., Xie T., Zhang L., Pei J., Mei H. “MAPO: Mining and Recommending API Usage Patterns.” In Proceedings of European Conference on Object-Oriented Programming (ECOOP), pp. 318-343, 2009.

[14] Wu, W., Khomh, F., Adams, B., Guhneuc, Y.G., Antoniol, G. “An exploratory study of api changes and usages based on apache and eclipse ecosystems.” In Journal of Empirical Software Engineering, vol. 21, pp. 23662412, 2015.

[15] Zhou, Y., Gu, R., Chen, T., Huang, Z., Panichella, S., Gall, H. “Analyzing APIs Documentation and Code to Detect Directive Defects.” In Proceedings of International Conference on Software Engineering (ICSE), pp. 27-37, 2017.

[16] Carter, E. “New Research Predicts Continued Growth in API Testing Market.” ProgrammableWeb (16-01-2018) Available: https://www.programmableweb.com/news/new-research-predicts-continued-growth-api-testing-market/brief/2018/01/16 Accessed [20-02-2019].

[17] Sohan, S. M., Anslow, C., Maurer, F. “A Case Study of Web API Evolution.” In Proceedings of IEEE World Congress on Services (ICWS), pp. 245-252, 2015.

[18] Bertram, D., Voida, A., Greenberg, S., Walker, R. “Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams.” In Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW), pp. 291-300, 2010.

[19] Uddin, G., Robillard, M. P. “How API Documentation Fails.” In Journal IEEE Software, vol. 32, pp. 68-75, 2015.

[20] Robbes, R., Lungu, M., Rthlisberger, D. “How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem.” In Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT/FSE), pp. 56, 2012.