A Data-Driven Approach to Prescribe Web API Evolution

Rediana Koçi

UNIVERSITAT POLITÈCNICA DE CATALUNYA
October 2023

A Data-Driven Approach to Prescribe Web API Evolution

Ph.D. Dissertation

Rediana Koçi

Thesis submitted:	October, 2023
Main Ph.D. Supervisors:	Prof. Xavier Franch
	Prof. Petar Jovanovic
	Universitat Politècnica de Catalunya, BarcelonaTech, Spain

Thesis submitted:	October, 2023
Main Ph.D. Supervisors:	Prof. Xavier Franch
	Prof. Petar Jovanovic
	Universitat Politècnica de Catalunya, BarcelonaTech, Spain
PhD Committee:	Prof. Antonio Ruiz Cortés
	University of Sevilla, Seville, Spain
	Prof. Cesare Pautasso
	Università della Svizzera Italiana (USI), Lugano, Switzerland
	Prof. Silverio Martínez-Fernández
	Universitat Politècnica de Catalunya, BarcelonaTech, Spain
PhD Series:	Barcelona School of Informatics, Universitat Politècnica de Catalunya, BarcelonaTech

Abstract

In the last two decades, the use of web Application Programming Interfaces (APIs) has grown exponentially, providing both consumers (software developers) and providers (companies and institutions that expose their organizational data or services) with numerous advantages and facilities. Web APIs allow developers to easily integrate existing services or data into their applications, reducing the time and cost required to build new software. At the same time, by making available their web APIs, providers can increase their customer reach or create a new revenue stream by monetizing the web API.

Even though consumers expect web APIs to be steady and well-established, web APIs are prone to continuous changes, evolving several times through their lifecycle. As the use of web APIs continues to grow, the need to evolve them to meet the changing needs of consumers becomes more challenging. However, consumers do not always welcome web API evolution, as the changes might be too frequent or not relevant from their point of view. This discontentment becomes even stronger for the fact that web APIs are exposed over the Internet, meaning that consumers may be forced to upgrade to new web API versions (if providers decide to discontinue the former ones). Knowing the impact changes have on consumers, providers have to strike a balance between not imposing unexpected, frequent changes and providing an up-to-date, maintainable, bug-free web API, that fulfills consumers' needs.

The aim of this research work is to develop a semi-automatic method that enables web API providers to prescribe changes based on their consumers' behavior, as recorded in web API usage logs. We take the position that web API evolution should be mainly usage-based, i.e., the way consumers use web APIs should be one of the main drivers of web API changes. We start with the exploration of web API evolution, by seeing the process from the points of view of both providers and consumers. We identify and classify the changes that often happen to web APIs, and investigate how all these changes are reflected in various artifacts. Building on this classification, we examine different types of usage logs, namely development and production logs, to

understand consumers' needs based on their behavior. We detect usage patterns that indicate the presence of usability issues or needs for improvement in the web API and suggest changes that should be implemented in future web API releases. Our approach enables providers to make informed decisions based on usage patterns. By adopting a usage-based approach, providers can ensure that their web APIs continue to meet the evolving needs of their consumers.

Resumé

En les últimes dues dècades, l'ús d'Aplicacions de Programació d'Interfícies (APIs) web ha crescut exponencialment, oferint nombroses avantatges i facilitats tant als consumidors (desenvolupadors de programari) com als proveïdors (empreses i institucions que exposen les seves dades o serveis organitzatius). Les APIs web permeten als desenvolupadors integrar fàcilment serveis o dades existents en les seves aplicacions, reduint el temps i el cost necessaris per construir nous programaris. Al mateix temps, al posar a disposició la seva API web, els proveïdors poden augmentar el nombre de clients de la seva marca o crear una nova font de guanys monetitzant la seva API web.

Tot i que els consumidors esperen que les APIs web siguin estables i ben establertes, les APIs web són propenses a canvis contínues, evolucionant diverses vegades al llarg del seu cicle de vida. A mesura que l'ús de les APIs web continua creixent, la necessitat de fer-les evolucionar per satisfer les necessitats canviant dels consumidors esdevé més desafiadora. No obstant això, els consumidors no sempre acullen amb satisfacció l'evolució de les APIs web, ja que els canvis poden ser massa freqüents o no rellevants des del seu punt de vista. Aquesta descontentament esdevé encara més fort pel fet que les APIs web estan exposades a través d'Internet, el que significa que els consumidors poden ser obligats a actualitzar-se a noves versions de la API web (si els proveïdors decideixen discontinuar les antigues). Coneixent l'impacte que els canvis tenen en els consumidors, els proveïdors han de trobar un equilibri entre no imposar canvis inesperats i freqüents i proporcionar una API web actualitzada, mantenible i sense errors, que satisfaci les necessitats dels consumidors.

L'objectiu d'aquest treball de recerca és desenvolupar un mètode semi-automàtic que permeti als proveïdors de web API prescriure canvis basats en el comportament dels seus consumidors, tal com es registra en els logs d'ús de la web API. Considerem que l'evolució de la web API ha de ser principalment basada en l'ús, és a dir, la manera com els consumidors utilitzen les web APIs ha de ser un dels principals impulsors dels canvis en la web API. Comencem amb l'exploració de l'evolució de la web API, veient el procés des dels punts de vista tant dels proveïdors com dels consumidors de la API. Identifiquem i classifiquem els canvis que sovint es produeixen en les web APIs, i investiguem com aquests canvis es reflecteixen en diversos artefactes. Analitzem diferents tipus de logs d'ús, concretament els logs de desenvolupament i producció, per entendre les necessitats dels consumidors basades en el seu comportament. Identificant les necessitats dels consumidors, podem prescriure canvis que s'haurien d'implementar en futures versions de la web API. El nostre enfocament permet als proveïdors prendre decisions informades basades en els patrons d'ús. Adoptant un enfocament basat en l'ús, els proveïdors poden garantir que les seves web APIs continuïn satisfent les necessitats en evolució dels seus consumidors.

Acknowledgements

I am taking the opportunity to thank all the persons without whom this thesis would not have been possible.

First, I want to thank my two advisors, Dr. Xavier Franch and Dr. Petar Jovanovic for their constant guidance and support throughout my PhD. Thank you for all the effort and time dedicated to our weekly meetings, and the patience you demonstrated during these years. I would also like to extend my gratitude to Dr. Alberto Abello. Though not an advisor 'de jure', he has been there since day one, and with his patience and prudence helped me overcome even the most challenging moments. I am really grateful and honored to have been your PhD student. I cannot imagine my PhD journey without the help and wisdom of each one of you.

Additionally, I would like to thank all the members of the DTIM and GESSI groups for the healthy and inspiring work environment. Thank you for all the shared experiences, discussions, and suggestions, mainly during our lunch seminars, which have been one of the most interesting and productive activities within the group.

I want to also thank all my friends, both here in Barcelona and abroad. To those here, thanks for the unforgettable moments we have shared while exploring and devouring this amazing city, which we've been lucky to call home for these years. To my friends abroad, thank you for always managing to brighten my days with your calls, messages, and a lot of funny memes even though miles away. I want to especially thank Soti for being so considerate and patient during this long journey of mine (ours :)), even when it was challenging for me to be the same.

Last but not least, I want to thank my family for always believing in me, and for their constant support and encouragement in all my pursuits.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1 Motivation

Since their first release over two decades ago (i.e., SOAP in 1998 [98] and REST in 2000 [28]), web APIs have been rapidly adopted by developers. Furthermore, the availability and usage of web APIs have significantly increased in recent years. According to the RapidAPI 2022 annual report [74], more than 62.6% of developers relied on APIs more in 2022 compared to 2021, with 69.2% expecting to rely on APIs even more in 2023. Many companies and institutions have adopted API-driven strategies to create new revenue streams and gain a competitive advantage by accelerating market entry. Businesses from several domains, like healthcare (e.g., openFDA, DHIS2), transportation (e.g., Uber, Barcelona TMB), travel (e.g., Skyscanner, Amadeus), and education (e.g., Moodle, Mendeley) are API-fying their activities and exposing their data through APIs. Developers, in turn, have become increasingly reliant on web APIs when building applications, which poses new challenges for web API providers, particularly during web API evolution, as evolving and maintaining large web APIs with numerous consumers becomes arduous.

Web APIs work as a contract between consumers' applications and providers' data, features, or services. Therefore, changes in these contracts can have a significant impact on these applications, causing a chain of modifications that are hard to manage. At the same time, providers are compelled to perform changes to keep their web APIs updated with the last business requirements, changing technology, and demanding consumers. As many real-world cases prove, the current evolution practices are unsuitable for data-intensive web APIs (i.e., smart APIs) [3]. These methods are neither scalable nor time-efficient [75]. To tackle these issues, providers must adopt efficient and effective evolution practices to manage their resources better and develop improved future versions of their web APIs. With the number of web APIs growing rapidly, and consumer interest following the same trend, the evolution process of web APIs needs to evolve as well.

In order to help providers in better planning and deciding on the changes, we introduce a data-driven usage-oriented approach that assists providers in analyzing consumers' implicit feedback, understanding their behavior, identifying their needs, and anticipating changes that meet these needs. We propose the analysis of web API usage logs, which data providers already own, but do not proactively use to inform their evolution practices. These log data contain all the traces of the interaction between consumers and web APIs, e.g., the requests to web API endpoints, the parameters used in the requests, and the order in which endpoints are being called. Hence, their analysis can detect the need for improvement or changes in the part of web APIs consumers most struggle with or inefficiently use. For instance, if providers identify repetitive calls to a specific endpoint returning client-side errors, they may review the error message and make it more detailed, so consumers can understand where the problem in their request is, and fix it before resubmitting other requests. If providers detect endpoints that return large responses, they may consider adding new parameters to that endpoint to better filter the data, or reducing the size of pagination. These are two simple examples of how the information in the usage logs can reveal several issues, the fixing of which could significantly improve the web API.

By adopting a usage-driven approach to web API evolution, providers can manage their resources more effectively, build improved versions of their web APIs, and enhance the overall user experience for consumers. This way, evolution planning will be more manageable for providers, and more fitting for consumers.

2 Background

In this section, we describe two main concepts we broadly refer to in the remainder of the thesis. First, we give a general overview of web APIs, how they work, how they differ from traditional APIs and their life cycle. Second, we introduce the concept of usage logs, as the main input of our approach. As chapter 4 is entirely dedicated to web API usage logs, here we just introduce the main concept, needed to easily grasp each chapter of the thesis. In the end, we introduce the web APIs we have taken into the studies throughout our work in several papers.

2.1 Web APIs

Web APIs are interfaces that allow different software to interact and communicate with each other over the Internet. Applications can request and receive data or functionality from other applications through these standardized interfaces. Thus, developers are increasingly relying on web APIs when creating their applications. They can use web APIs to retrieve data from a database, integrate with third-party services, automate tasks, or build new applications on top of existing ones. For instance, using Twitter web API, a developer can retrieve a user's tweets, post a new tweet, or search for tweets that contain a specific keyword.

Web API requests are typically sent using HTTP methods such as GET, POST, PUT, and DELETE, which allow clients to retrieve data, submit new data, update existing data, and delete data from the server, respectively.

While traditional APIs and web APIs are both used to enable communication between different applications, there are some key differences between them. This thesis focuses on the evolution of web APIs. As such, a comparison between the two types is necessary to point out these differences.

Traditional APIs (often referred to as library APIs or local APIs) are programming interfaces that serve as means of communication between two applications running on the same machine or server. These APIs typically use language-specific interfaces and data structures of the programming language in which they are written. Conversely, web APIs enable remote communication between applications running on different servers or different locations. Web APIs, as already stated, use standard web protocols such as HTTP and HTTPS, and typically communicate using a predefined set of data exchange formats such as JSON or XML.

The differences in the way these two types of APIs work give their respective consumers and providers advantages and disadvantages. While traditional APIs evolve, their consumers have more control over upgrading to the new versions. As long as they have the API locally stored in their machines, they can somehow delay the upgrade process. The opposite happens with web APIs. As consumers access them remotely, if providers decide to perform some changes or disconnect some older versions of the web API, consumers are forced to upgrade to the changes to prevent their applications from misbehaving or crashing. Thus, the evolution of web APIs becomes more onerous for consumers, and therefore more challenging for their providers.

2.2 Web APIs usage logs

According to the 8th Lehman's Law for software evolution, feedback is essential in the software evolution process [49, 50]. Gathering consumers' feedback is a followed practice by many web API providers, e.g., Twitter, Facebook, and Google, which regularly solicit feedback from their users using surveys, forums, issue trackers, and social media. While these ways of communicating with users are certainly important and such communication channels facilitate the interactions between providers and consumers, analyzing this feedback from different sources can be challenging. It involves sifting through a large volume of typically unstructured data from multiple channels and does not ensure that all the consumers' requests are regarded.

While the remote accessibility of web APIs poses new challenges and increases the responsibility of web API providers, they can also benefit from this by observing the behavior of their consumers. All requests sent to the web API endpoints can be tracked by providers. In fact, web API usage is regularly monitored by providers to ensure uninterrupted service and to generate several reports regarding web API traffic. However, web API usage logs possess even greater potential. They contain all the requests consumers send to web API, the parameters used, and the sequences of these requests to fulfill certain functionality. Thus, they can be considered implicit consumer feedback. In addition to being gathered and analyzed in a more transparent, centralized, and scalable way, these logs contain particular information that can not be collected directly from consumers. By analyzing the sequences of all requests, providers can identify potential areas for improvement in the endpoints, some of which may not have been evident to consumers. In addition, analyzing usage logs can reveal new web API usage scenarios that were not intentionally designed by providers but are nonetheless being called by consumers.

We distinguish two types of logs: (i) development logs, and (ii) production logs. Development logs are generated at design time, while developers build and test their applications. By analyzing these logs, we can identify usability issues (e.g., issues with clarity, memorability, and helpfulness) consumers face when they integrate the web API into their applications. We cover their analysis in Chapter 5. Production logs are generated during application runtime, while applications are being used by end users. By analyzing these logs, we can identify usage patterns (i.e., relationships between endpoints inferred from frequent sequences of calls), which characterize consumers' behavior and might be indications for change or improvement. We cover their analysis in Chapter 6.

2.3 Use Cases

To exemplify our approaches, we analyzed usage logs of two main web APIs, (i) District Health Information System web API (DHIS2), and (ii) Facultat d'Informàtica de Barcelona web API (FIB).

DHIS2 is an open-source, web-based platform serving as a health management information system. It is developed and maintained by the software development group within the Health Information Systems Programme at the University of Oslo (UiO), Department of Informatics. DHIS2 platform is used worldwide in more than 100 countries by various institutions and NGOs for data entry, data quality checks, and reporting. It has an open REST WAPI, used by more than 60 native applications (consumers). For the analysis, we use logs from two different DHIS2 instances: (1) the World Health Organization (WHO), in their Integrated Data Platform (WIDP), which is used by several WHO departments for routine disease surveillance and country reporting; (2) Médecins Sans Fontières (MSF), used for field data collection and as a central repository for medical data.

DHIS2 WAPI resources (e.g., data, documents, functionality) are exposed through WAPI endpoints (e.g., `api/dataValueSets)`. Using these endpoints, consumers can access and manipulate data stored in the instance of DHIS2, data related to disease cases (e.g., where a disease or infection spread, number or cases), organization units collecting the data (e.g., hospitals), etc. Consumers can interact with the WAPI using HTTP methods: call a GET request to retrieve a resource, a POST request to create one, PUT to update a resource, and DELETE to remove it, e.g., `GET api/events, POST api/dataSets/ID/version`.

The FIB WAPI was developed by the inLab research laboratory at the Polytechnic University of Catalonia (UPC). It provides a set of endpoints for extracting data about departments, courses, exams, room reservations, etc. It is a read-only WAPI, meaning that consumers can only retrieve the data using GET requests, e.g., `GET v2/assignatures, GET /v2/lectures`. It is built under REST architecture and is mainly being used by the FIB website, monitoring systems, school news screens, and several applications created for academic purposes.

Even though the number of web APIs we take in the study is limited to two, the web APIs we have chosen are from different domains (health and education) and of different types. The DHIS2 web API is publicly available and can be used by anyone, whereas the FIB web API is private and requires UPC account credentials for access. Our aim was to demonstrate that our approach can be applied to a variety of APIs, regardless of their type or domain.

3 State-of-the-art

Even though web API evolution has gained considerable attention, most of the available works analyze this process from the consumers' perspective. Thus, their most targeted objective is to settle the issues faced by consumers when upgrading to the new web API versions and to smooth this process (the upgrade process) as much as possible. While their contribution is invaluable and the developed tools and methods greatly improve consumers' migration to the new releases, they mostly suggest post-evolution practices. We refer to post-evolution practices as actions providers (should) take to make consumers' upgrade process easier and more straightforward after they have designed and implemented the changes. However, as Abelló et al. state in their work, "the core problem in web API evolution is to determine what, how, and when to evolve" [3].

Lübke et al. focused on API evolution strategies and introduced a set of patterns by mining real-world API documentation and conducting workshops with practitioners [54]. These patterns consist of defining API descriptions, introducing version identifiers, using semantic versioning, and also practices regarding the lifetime of older APIs when new ones are provided. Serbout and Pautasso also focused their work on identifying the current versioning practices used by web API providers to ease compatibility checking and maintainability for both consumers and providers [81]. Espinha et al. discourage frequent changes and highly recommend the application of blackout tests: disconnecting for short times the old versions that are planned to be removed so that consumers will be reminded of the upcoming changes [27]. Sohan et al. suggest the use of semantic versioning, improvement of documentation, and the use of effective communication channels to inform consumers about the changes [82]. In the same line stand the findings of Lamothe et al. in their systematic literature review of API evolution [47]. As they observed, the majority of the proposed tools and techniques aim to improve API usage, provide usage recommendations, help with migration, reduce API misuse, and create improved API documentation. Interestingly, they identified the mastering of feedback systems in API evolution as the least explored topic.

The importance of consumer feedback has been often tackled, but the current methods of gathering and analyzing this feedback are not effective, lack scalability, and have several limitations. Rauf et al. presented surveys, controlled experiments, and usability as the most used methods in evaluating API usability [75]. As the authors showed, there is a need for more automated, scalable, and time-efficient methods. Zhang et al. studied different ways followed by API designers to gather user feedback [102]. They found out that often API designers refer to informal channels to get users' feedback. They mentioned bug reports, emails, and online discussions as the primarily used way to identify usability issues reported by users. However, API designers participating in the study showed a strong interest in understanding users' mental models from low-level log data but expressed the lack of tools and difficulty in analyzing these data, considering their enormous volume.

Mathijssen et al. performed a systematic literature review on different API management practices applied by providers [60]. They showed that providers were logging the access to their API and monitoring the API usage, but with the aim of providing performance statistics or traffic metrics, and not using them as input for the API evolution planning. Ivanchikj et al. used web (REST) API logs to feed their RESTalk Miner tool [36]. This tool employs RESTalk, a domain-specific language, to create visual representations of API conversations [37, 69]. While it does have pattern-searching capabilities, its primary purpose is to facilitate the visualization of these conversations. Macvean et al. assessed the usability of web APIs by analyzing the data from Google API Explorer to identify APIs [56]. The metric they used to measure API usability was API request error rate (client-side erroneous requests (4XX) per total requests to the API). Although their results were still preliminary and, as they stated, early in nature, the methodology used seems promising and opens a lot of areas for future research.

While the importance of consumer feedback in web API evolution is acknowledged [3,32,46,47], gathering and efficiently analyzing it remains an open issue. Our approach contributes to this identified gap. We propose analyzing the consumers' implicit feedback, represented in web API usage logs. Hence, we suggest leveraging some valuable data providers already own, but do not use to their full potential. To this aim, first, we observe the evolution process from both consumers' and providers' perspectives. Secondly, we show the importance and the potential of the web API usage logs, as the main input of our approach. Then, we thoroughly analyze the usage logs, applying different techniques (i.e., statistical analysis, process mining), to detect different patterns, whose occurrence might indicate the need for potential changes. This way, providers can use more practical and scalable methods to plan the changes they will implement. Moreover, evolution will better meet consumers' needs and requirements, as their usage was the main driver of the changes.

4 Thesis Objectives and Research Questions

The goal of this research work is to propose a semi-automatic data-driven approach for web API providers to prescribe web API changes by studying their consumers' behavior, imprinted in web API usage logs. Figure 1.1 depicts the main steps of our methodology.

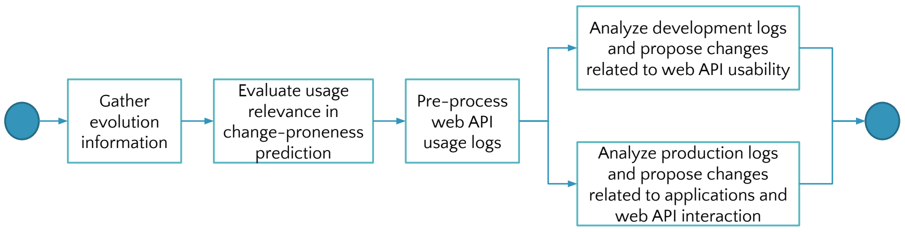

Gather evolution information
Evaluate usage relevance in change-proneness prediction
Pre-process web API usage logs
Analyze development logs and propose changes related to web API usability
Analyze production logs and propose changes related to applications and web API interaction

Fig. 1.1: Overview of the proposed methodology.

We will approach our goal by answering the following research questions. Each question corresponds to one step in the introduced methodology (Figure 1.1).

- RQ1: Which are the changes that happen to web APIs when they evolve?
  - RQ1.1: How are the changes that happen to web APIs reflected in different web API artifacts?
  - RQ1.2: Which are the causes of the web API changes?
  - RQ1.3: To what extent are the web API changes reflected in the usage logs?
- RQ2: To what extent can we predict the change-proneness of web API endpoints?
  - RQ2.1: Which metrics can be used for the change-proneness analysis of web API endpoints?
  - RQ2.2: To what extent can usage metrics improve the change-proneness prediction of web API endpoints?
  - RQ2.3: Which characteristics help to determine the web API endpoints that are more likely to change?
- RQ3: How can web API usage logging be improved?

  - RQ3.1: What are the advantages of proactively analyzing web API usage logs?
  - RQ3.2: Which are the main impediments to using web API usage logs regarding consumers' behavior?

- RQ4: Which web API usability changes can be anticipated based on developers' behavior in the web API usage development logs?
  - RQ4.1: Which are the usability sub-attributes that mostly influence the API consumers' experience?
  - RQ4.2: Can we find issues impacting these usability sub-attributes by analyzing the web API usage logs?
  - RQ4.3: Can the usability issues found in the web API usage logs be measured in a meaningful way from the API consumers' point of view?

- RQ5: Which web API changes can be anticipated based on applications' behavior?
  - RQ5.1: In which ways can endpoints be related to each other to indicate the need for change based on how consumers call them?
  - RQ5.2: What information can we extract from usage logs, that can help in detecting the defined relationships between endpoints?
  - RQ5.3: How can we detect the occurrence of the patterns in the usage logs?
  - RQ5.4: To what extent can we propose changes based on the patterns found in the usage logs?

5 Contributions

The contributions of this research work are fourfold. They are related to the methodology steps introduced in Figure 1.1, and derive from answering the above-mentioned research questions:

- *Classify the changes at the interface level that typically happen to web APIs when they evolve.* To have a complete set of changes, we analyze different artifacts used by providers during their web API evolution, as well as artifacts used by them to communicate the implemented changes. It proved to us that most of the studied artifacts contained incomplete information, lacking the majority of the newly introduced changes. Due to their incompleteness, we extracted the set of changes from the web API controller code by comparing consecutive versions, which we then considered as the ground truth.

- *Show the relevance of usage in web API change-proneness prediction.* We take the position that web API evolution should be driven by how consumers use web APIs. Thus, to prove our approach, we included usage-related metrics in change-proneness prediction models. To date, change-proneness was mostly analyzed based on design metrics and previous changes in the artifacts. Our experiments showed that incorporating usage metrics into the prediction model not only improved accuracy but also enabled early prediction of changes that providers implement at later stages. By considering both design and usage metrics, API providers can make proactive decisions, ensuring that their changes align with user expectations.

- *Unveil the potential as well as the main impediments of proactively using web API usage logs in the context of web API evolution.* Web API usage logs contain invaluable information regarding consumers' behavior, needs, and difficulties, albeit requiring tedious pre-processing. To leverage this potential, we propose several purpose-driven analyses that can be applied to these logs. Furthermore, driven by the analysis requirements, we identify potential issues related to the format and the logging of these data and provide suggestions for mitigating them.

- *Provide a systematic, data-driven method to measure the usability of web APIs based on web API usage logs.* We conducted an empirical study to measure web API usability by monitoring and analyzing web API usage logs. For each usability attribute extracted from the literature review, we defined a set of indicators and their respective metrics. We built a classification model to predict consumers' error rate per endpoint, based on the defined metrics. Our model was able to predict the class of the endpoints (endpoints with high or low usability) with an accuracy of 72.25%.

- *Provide a set of usage patterns, whose occurrence in the web API usage logs may indicate the need for future change.* We propose a process mining-based method to anticipate changes in web APIs, by analyzing the behavior of web API consumers and identifying their needs in the web API usage logs. We start by defining a set of usage patterns, each of which represents a specific consumer behavior, and indicates the need for a change in web API. The detected instances of the patterns in the two web APIs we took in the study, proved to be significant for the consumers and the providers of the studied web APIs.

6 Structure of the Thesis

This research work is structured in five main chapters of this document (i.e., Chapter 2 - Chapter 7). Each chapter is self-contained, corresponding to an individual research paper, thus it can be read in isolation. Even though the terminology used has been clearly defined in each particular chapter, it is worth mentioning that for the concept of web API, we interchangeably use web API or the acronym WAPI.

The papers included in this thesis are listed below. Chapter 2 is based on Paper 1, Chapter 3 is based on Paper 2, Chapter 4 is based on Paper 3, Chapter 5 is based on Paper 4, Chapter 6 is based on Paper 5, Chapter 7 is based on Paper 6, and Appendix A is based on Paper 5.

P1. Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. "Classification of changes in API evolution." In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC),* pp. 243-249. IEEE, 2019 [Short paper].

P2. Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. "Web API Change-Proneness Prediction." [Submitted].

P3. Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. "Improving Web API Usage Logging." In: *Research Challenges in Information Science: 15th International Conference, RCIS 2021, Limassol, Cyprus, May 11–14, 2021, Proceedings,* pp. 623-629. Cham: Springer International Publishing, 2021 [Poster paper].

P4. Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. "A data-driven approach to measure the usability of web APIs." In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA),* pp. 64-71. IEEE, 2020.

P5. Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. "Web API evolution patterns: A usage-driven approach." *Journal of Systems and Software (2023)*: 111609.

P6. Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. "PatternLens: Inferring evolutive patterns from web API usage logs." In: *Intelligent Information Systems: CAiSE Forum 2021, Melbourne, VIC, Australia, June 28–July 2,* *2021, Proceedings*, pp. 146-153. Cham: Springer International Publishing, 2021 [Demo paper].

6.1 Classification of changes in API evolution (RQ1)

Following the issues pointed out in the introduction and the identifying gap in the State of the Art, we saw it reasonable and necessary to start our research work by observing the web API evolution process from the planning, on the providers' side, to the changes adaptation on consumers' side. Figure 1.2 depicts the main steps we followed. We perform an exploratory analysis of web API evolution, including the identification of commonly implemented changes, their documentation, and consumers' compliance with the changes. We identify and classify the set of possible changes frequently happening on web APIs. In the subsequent sections, we aim to identify the need for these changes, by considering the defined set of changes as the ground truth. The analyzed artifacts are illustrated in Figure 1.3.

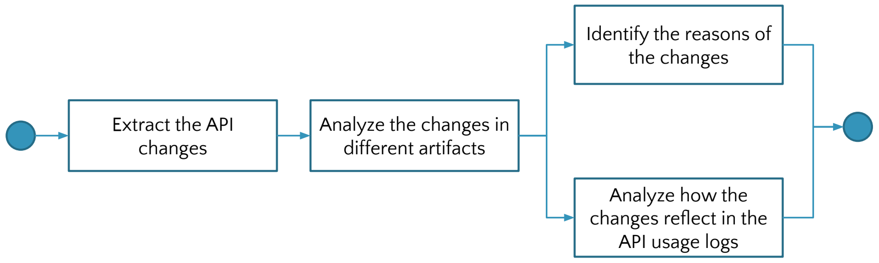

Extract the API changes
Analyze the changes in different artifacts
Identify the reasons of the changes
Analyze how the changes reflect in the API usage logs

Fig. 1.2: Overview of Chapter 2.

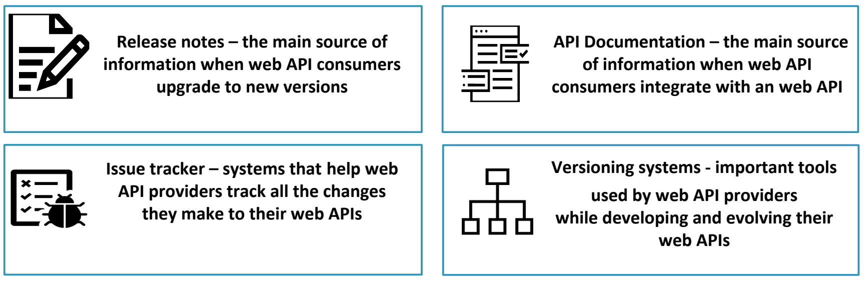

Release notes – the main source of information when web API consumers upgrade to new versions
API Documentation – the main source of information when web API consumers integrate with an web API
Issue tracker – systems that help web API providers track all the changes they make to their web APIs
Versioning systems - important tools used by web API providers while developing and evolving their web APIs

Fig. 1.3: API artifacts.

6.2 Web API Change-Proneness Prediction (RQ2)

Next, we analyze the relevance of consumers' behavior (stored in web API usage logs), in change-proneness prediction. Typically, change-proneness is associated with design characteristics and the history of changes in the artifacts. However, providers implement changes for several other reasons, including consumers' feedback. While gathering, processing, and analyzing consumers' explicit feedback can be time-consuming, implicit feedback in the form of usage logs remains largely unexplored. These logs offer a more systematic means of gathering and analyzing feedback. Consequently, we propose a methodology for predicting the change-proneness of web API endpoints, taking into account not only design and change history but also their usage. We introduce a set of metrics covering the above-mentioned characteristics and evaluate our approach on a real-world web API. Our preliminary findings indicate that incorporating usage metrics into the prediction model not only improves accuracy but also enables early prediction of changes that providers implement at later stages. By considering both design and usage metrics, API providers can make proactive decisions, ensuring that their changes align with user expectations.

6.3 Improving Web API Usage Logging (RQ3)

As usage logs represent the main input of our approach, we specifically stop on their potential for evolution purposes, and how useful and helpful they can be if fully and properly analyzed. We start by making a distinction between two types of logs (as already mentioned in Section 2.2), namely development and production logs. Even though the same format, these logs can be used to uncover different issues in web APIs as they are generated at different moments of consumers' application lifecycle. We suggest several methods that can be applied to each of them (e.g., process mining), mention purpose-driven analyses (e.g., to analyze the usability of web API, to understand consumers' needs, to detect usage patterns), and also introduce the main issues and challenges related to the format of these logs. As these logs are not logged for these kinds of analyses or to target these goals or purposes, they lack some details and/or features. Figure 1.4 shows the two main challenges in web API usage logs pre-processing: (i) field extraction, and (ii) session identification. We suggest how to overcome these challenges in their preparation and pre-processing, and also suggest some practical guidelines on how to better log this information.

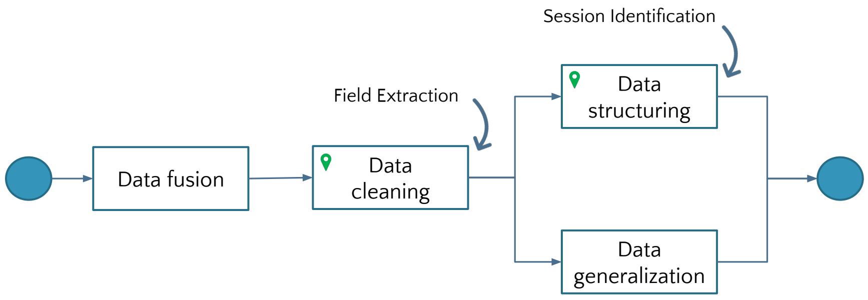


Fig. 1.4: Overview of Chapter 4.

6.4 A data-driven approach to measure the usability of web APIs (RQ4)

After classifying and exploring usage logs, we start analyzing them. The goal is to analyze the usability of web APIs and investigate how the main usability issues can be foreseen by reading between the usage log lines (entries). We review the current state of the art in web API usability, and for the gathered usability attributes, we define indicators that could be quantified with the available information in the web API usage logs. For instance, for the *clarity* usability sub-attribute, we define the similarity of web API elements' names as an indicator of *clarity* and quantify it in the average similarity and maximum similarity metrics. Assuming that a web API endpoint that suffers from poor clarity will have a high error rate (erroneous request per all requests), we detect that endpoints, whose path elements' names have a high similarity, have also a high error rate, thus lacking clarity. Figure 1.5 depicts the main steps we followed in defining the usability attributes, indicators, and metrics.

6.5 Web API evolution patterns: A usage-driven approach (RQ5).

We define a set of web API behavioral patterns, whose occurrences in the usage logs indicate the need for changes. To detect the instances of the defined patterns, we introduce several metrics following process mining techniques. To show the applicability of our approach, we apply it to usage logs from two different web APIs. Moreover, to show the significance of the detected patterns, we interview the consumers and the providers of the studied web APIs. Figure 1.6 depicts the main steps we followed to detect the patterns and suggest the changes they implied.

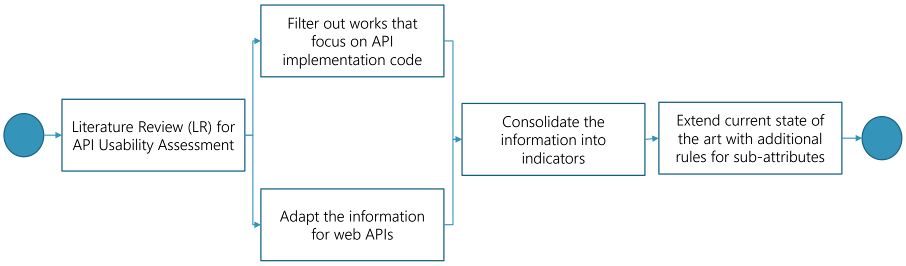

Literature Review (LR) for API Usability Assessment
Filter out works that focus on API implementation code
Adapt the information for web APIs
Consolidate the information into indicators
Extend current state of the art with additional rules for sub-attributes

Fig. 1.5: Overview of Chapter 5.

6.6 PatternLens: Inferring evolutive patterns from web API usage logs (RQ5).

To exemplify the approach introduced in Chapter 6, we build PatternLens, a tool that takes as input the web API usage logs, detects the occurrences of the patterns (pre-defined in the tool), and introduces them to providers, along with the suggested changes. Providers can accept or reject the suggested patterns,

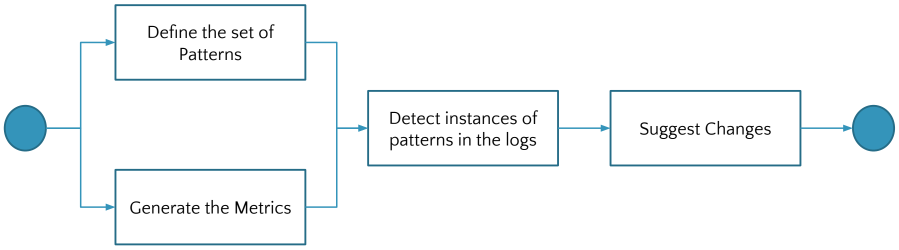

Define the set of Patterns
Generate the Metrics
Detect instances of patterns in the logs
Suggest Changes

Fig. 1.6: Overview of Chapter 6.

Chapter 2

Classification of Changes in API Evolution

The paper has been published as a short paper in the *Proceedings of the IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC).*
DOI: http://dx.doi.org/10.1109/EDOC.2019.00037

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 the World Health Organization (WHO).

1 Introduction

Nowadays, Application Programming Interfaces (APIs) are being broadly used [17]. The main reason for this success is that APIs provide advantages to both 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 by 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 consumers could be described as follows: API producers develop a stable API, providing very detailed and helpful documentation, so 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 backward incompatible and supported by poor documentation. All of this has a negative impact on consumers [22, 26, 53]. Facing a lot of difficulties when upgrading to new versions, or even having a bad experience with how the evolution process happens and is communicated to them, consumers see the evolution as a set of painful changes, rather than a necessary phase bringing them benefits.

API producers have their own reasons when performing evolutive actions, being forced to make changes to their APIs in order to add new features, make them simpler, improve their maintainability, fix bugs, optimize their performance, or improve their security [15, 22, 101]. 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 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 kinds of changes, for Patch and Minor versions, are non-breaking changes. They will not prevent existing consumers' applications from functioning after the upgrade, while consumers can optionally learn and modify their code to benefit from new features and improvements. Conversely, the last changes, introduced in Major versions, are breaking changes. After upgrading to Major versions, consumers are required to modify their code to comply with the new API version. Thus, API producers should avoid these kinds of changes as much as possible.

The cost of adapting to the new changes depends on the type of changes and on

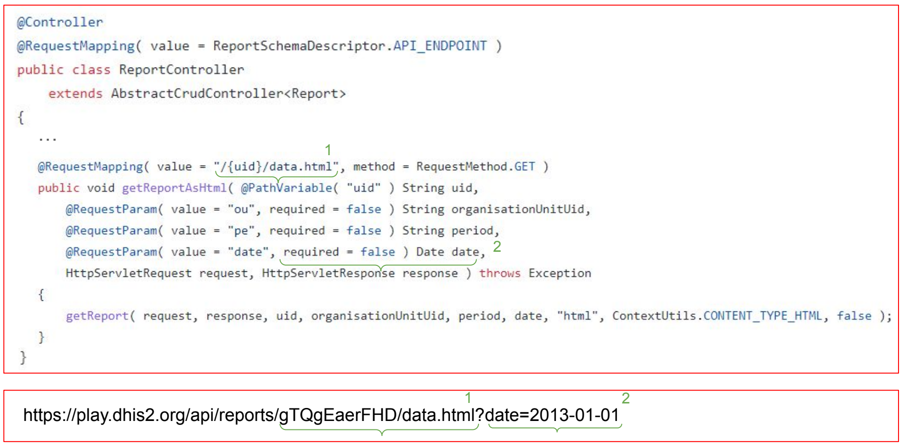
[1]https://semver.org

the usage of the changed API elements: how much are the changed elements used and how do the changes directly affect the API calls? The upgrade burden can be reduced by supporting each release with more detailed documentation (e.g., release and upgrade notes).

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 first 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 to 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 [2, 95]. On the other hand, we take into study the issue tracker and the versioning system as two important tools used by API producers while developing and evolving APIs [13, 35]. We evaluate the impact these changes have on consumers by analyzing the API log files, as they contain the calls that consumers make to the API. This way we avoid analyzing the code of 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 to a real-world use case. Analyzing the complete API evolution lifecycle, from raising the issue, through its implementation, and documentation, publishing it in the release notes, and finally analyzing its usage through 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 API (API 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 build, 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



```
@Controller
@RequestMapping( value = ReportSchemaDescriptor.API_ENDPOINT )
public class ReportController
    extends AbstractCrudController<Report>
{
  ...
  @RequestMapping( value = "/{uid}/data.html", method = RequestMethod.GET )
  public void getReportAsHtml( @PathVariable( "uid" ) String uid,
      @RequestParam( value = "ou", required = false ) String organisationUnitUid,
      @RequestParam( value = "pe", required = false ) String period,
      @RequestParam( value = "date", required = false ) Date date,
      HttpServletRequest request, HttpServletResponse response ) throws Exception
  {
      getReport( request, response, uid, organisationUnitUid, period, date, "html", ContextUtils.CONTENT_TYPE_HTML, false );
  }
}
```

```
https://play.dhis2.org/api/reports/gTQgEaerFHD/data.html?date=2013-01-01
```

Cases 1 and 2 show that the API controller contains the specifications of API elements (names, hierarchies, etc.), that consumers need in order to make calls to API.

Fig. 2.1: API syntax in Controller and calls.

about API 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. Section 2 gives an overview of the state of the art of API evolution. In section 3, we present the classifications derived from a literature review. In section

4, we describe our methodology and apply it to a real-world use case. Section 5 discusses our findings, while section 6 concludes the paper and discusses future work.

2 Related Work

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

2.1 API evolution

The evolution of APIs has gained considerable attention from researchers recently covering different aspects of this cumbersome process [22, 53, 82, 97, 100]. Studies have been conducted to identify the changes that occur to APIs from older to newer versions [22, 97]. Wang et al. [97] gave a catalog 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 [22] manually identified the changes in 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' updates. Li et al. [53] 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 into analyzing the impact that API changes have on consumers' applications. Robbes et al. [76] assessed the impact of API deprecation on a Smalltalk ecosystem in terms of frequency, magnitude, duration, adaptation, and consistency of the ripple effect (adaption to API deprecation). Espinha et al. [26, 27] 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 a decrease in the satisfaction of Web APIs consumers. Thus, they recommended not changing the APIs too often and performing blackout tests (shutdown of the older versions of API in a short time frame).

Most of the above works pay attention to the consumer's side, leaving aside the API producer's side. However, the latter also face challenges and difficulties in managing and evolving API. Xavier et al. [101] made a survey to reveal the reasons why producers break APIs. Brito et al. [15] did a reason-based classification of changes in the APIs of 400 libraries. In both of the above-mentioned studies, they asked directly the API developers that made each change, 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 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, which 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 from one version to another, and a set 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 better understand the impact that changes have on consumers.

2.2 API usage

Different studies exist in analyzing API usage, i.e. the ways consumers use the API. Ed-douibi et al. [24] 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. [103] 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. [100] 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' application 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 the study. Each of them might have its 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' behavior from the API usage logs files. These files contain the calls of API consumers to the API. Combining knowledge of API evolution and its impact on API usage will be beneficial in understanding the overall API evolution process.

3 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. It is important and effective to look at and analyze the history of these changes in order to assist and anticipate the further evolution of APIs.

3.1 Which are the changes that happen to APIs?

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

From the consumers' point of view, referring to the compatibility of the new version with the previous ones, API changes can be divided into two types: breaking changes and not breaking changes [22]. Next, breaking changes can be classified as changes that affect the behavior of the API (pre and post-conditions, changes in API response) or the syntax of the API.

From the producers' 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 in attributes, methods, or classes) or the actions performed on them (e.g., moving elements, adding new ones, or refactoring). These classifications point out which parts of APIs are more stable and which ones are more change prone during their lifecycle. Brito et al. [15] used two first classifications in their work. They referred to changes in different API elements (types, methods, and fields) and the causes of the changes. Sohan et al. [82] 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's 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 kinds of changes, we suggest the comparison of the API controller of two consecutive versions. The only drawback of this option is that the API controller is available only for open-source projects. However, considering that the information provided from other sources (release notes, documentation, issue tracker, etc.) can be incomplete or inaccurate, from a consumer point of view, this is the only way to have the full set of changes that can be considered as ground truth. 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 "/", different resources can be accessed for a given value) or query parameters (part of the request body after '?' in a key=value form). Parameters can be required or optional. New parameters can be added to resources, existing ones can be removed, renamed, or 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 changing the needed authority.

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

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

While performing different 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. As in API evolution, there are no standards in any of these activities, there is no easy way to extract the changes from them. It highly depends on the project conventions and producers' way of coding and documenting.

We refer overall to four different API artifacts, namely release notes, API documentation, issue tracker, and versioning systems, to see how evolution is reflected in them and to what extent they document the changes.

- **Release notes** are documents that accompany the release of new software versions. These notes are used to communicate to software consumers the changes that happened to the newly issued version, such as the fixed bugs, the added features, improvements and extensions of existing features, and other changes that affect the API consumers' applications. Usually, these notes are manually written. Even though they play a crucial role in the upgrade process, there are no standard rules in writing them. Abebe et al. [2] 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 the issue tracker, size of issue description, number of days to address the issue, experience of issue reporter, etc [2]. As we can see, sometimes the likelihood of an issue being in the release notes depends on factors with no relevance for the consumers (e.g., length of issue title, reporter experience).
- **API Documentation** has a technical nature. Its primary role is to instruct users on how to use and integrate with the API. They provide detailed information about API elements, such as endpoints, resources, fields, types, and parameters, often showing examples [95]. API documentation is usually manually written, and sometimes this process is not synchronized with the new version releases. All these results in outdated documentation, which means that API changes are not timely reflected in it [104]. Actually, obsoleteness in documentation is one of the most important concerns of API consumers when they upgrade to new API versions [26, 27, 90]. Uddin and Robillard [90] suggest not expecting 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. They serve as a communication means between different actors of a project like managers, developers, and customers. Bertram et al. [13] 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 for the issue opening is not specified in the description, referring to comments can give a better understanding of it.

Developers open issues in the system when they create a new feature, improve an existing one, or fix a bug in their software. Customers can also report bugs or make requests for new features. When opening a new issue, different fields have to be filled: a short description, type of issue (bug, feature, improvement/enhancement), the version/s the issue is related to, component/s affected, severity, etc. Sometimes different fields are skipped or not properly filled, often by non-technical users of the systems [7,13]. This makes it difficult to later query information about the issues related to a specific version or component. This means that even if we query from the issue tracker systems all the issues related to API (information under `Component` field), for a specific version, the list generated can be incomplete, and will not provide us all the changes performed for that version. Besides this, issue tracker systems are not always accessible. They are open only in the case of open-source APIs.

- **Versioning systems**. The history logs of versioning systems contain information for every change in the repository. Developers can associate comments to their commits on versioning systems to explain what they did in the API. These comments can vary from simple and short descriptions to more detailed ones. It depends on the developers' style of coding and also project regulation. Hattori and Lanza [35] used a set of keywords to classify the changes performed based on words used in comments. For example, if the comments contained words (or parts of words) like *clean%*, *integrat%*, *migrat%*, or *polish%*, they classified the changes as maintenance activities, or management related. If the comments contained words like *implement%*, *add%*, *creat%*, *start%*, or *includ%*, they classified them as development activities or forward engineering. Similar to the issue tracker, versioning systems can be accessed only for open-source API.

3.3 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 [15,22,101]. The identification of the causes of the changes completes the big picture of the evolution of API.

Brito et al. [15] in their empirical study classified the breaking changes based on API producers' reasons to make these changes. According to their work, to add new features, producers move classes between packages, rename methods and perform changes in the parameter list. To simplify the API, to make it easier to use and with fewer elements, they remove classes, add final modifiers and perform changes in the parameter list. They usually move methods, rename methods, and move classes in order to improve the maintainability of APIs.

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 natures (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. Every new feature is related to 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 the API lifecycle. We adopted the usability taxonomy developed by Mosqueira-Rey et al. [64] to classify changes based on the target usability aspect of the API they aim to change (i.e., to improve or add functionalities), 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 the system and increase the interest of consumers in using it.

3.4 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 on either the server side (provider) or the consumer side. A log file contains different log entries. Each log entry represents a call to an API endpoint. An API call contains a lot of useful information about the protocol consumers used, the hostname of the API, base paths, relative paths, and query parameters, as in Fig. 2.2.

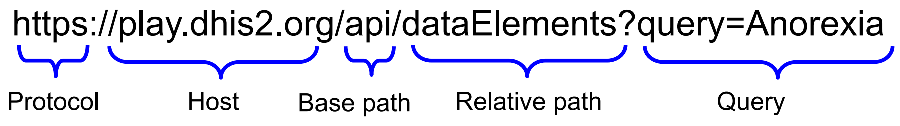


Fig. 2.2: An URL to call an API.

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 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. API providers can change the name of a resource, and its parameter list, move them up or down in the API elements hierarchy, can deprecate or even delete them. Consumers' effort to update with the changes in resources depends on how much they use them and of course on the type of changes. The query part of the call, in the form of key-value, contains parameters of the resource. Sometimes these parameters are optional, and consumers use them when they want to apply specific filters to the resources or to reduce the response size. Parameters can also be mandatory: consumers should include them in the call to access the desired resource. Thus, changes in the level of the parameters also affect the consumers. As described above, when the syntax of API changes, consumers have to make changes to the API calls. But, they may need to update their applications even when API changes impact only the behavior of the API and not the syntax. These changes in behavior can be related to the pre-condition, post-condition, or response of the API. Even though not directly, these changes can also appear in the usage logs, affecting the time to respond to the request, the size of the object returned, etc.

4 Use case: DHIS2 API

4.1 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 as well as the data from our use case.

First of all, we identify the changes by comparing two consecutive versions of API. Then we analyze different API artifacts, to see to what extent are the changes documented in them. We do this step manually, and for the sake of completeness, we refer to different sources. With the information provided in these artifacts, we classify the changes based on their causes. At the same time, we analyze the API usage logs to see the impact that these API changes have on consumers. In 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 emphasizes the impact each class of changes has on the consumers' side, providing us clues on changes that can be identified from the logs.

4.2 DHIS2 use case.

We applied our approach to 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 the 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 the WHO installation.

4.2.1 Which are the changes that happen to APIs?

We compared the API controller code of versions 2.26 and 2.27 in order to get the whole set of changes between them. 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. We used Beyond Compare[3], a software that provides a comparison of directories and different file formats (e.g., text, mp3, image).

[2]https://www.dhis2.org/about
[3]https://www.scootersoftware.com

Table 2.1: 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**

Overall, we found 38 changes introduced in the new API release, belonging to six different types of changes (Table 2.1).

- 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 introduced 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 the 2.27 version. API producers claimed to have removed not used endpoints or endpoints whose functionality was already replaced in the previous versions.
- 2 new authorities 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, the POST method was supported in the new version.

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

- **Release notes**: We analyzed the release notes for the release of version 2.27 of the DHIS2 system. The release notes were organized into different sections, each of which covers 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 in the release notes, they provided additional information in the form of demo examples, screenshots, links to issues raised in the issue tracker system (JIRA), and links to documentation, with more detailed information and explanations.

We analyzed the Web API Features section of the release notes[4]. 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 is not too detailed and lacks an explanation of how to use the new endpoint, its attributes, etc. Beside this, only six changes were introduced in the release notes.

- **API Documentation**: In order to see how changes are reflected in the API documentation, we compared the documentation of versions 2.26 and 2.27. We noted that the documentation sometimes was 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[5], it was documented for the first time in 2.27 documentation. There were also changes not yet documented like the removal of `ProgramStage-DataElements` endpoint. It has completely disappeared in version 2.27, and this was documented nowhere[6]. On the other hand, we saw that changes that appeared in the documentation were explained in detail. For example, for the new endpoint `deletedObjects`, a whole paragraph was added in the documentation, describing how it works, and giving examples of calls to the new endpoint.

- **Issue tracker**: DHIS2 uses JIRA as issue tracker system. We extracted all the issues of the 2.27 release from it. We filtered the issues based on issue type (Bug, Feature, Design, Epic, Test, User Story), fixVersion (all releases), and Component (Application components, API components, Test, Documentation, Frontend, Backend). JIRA provides more fields that we could manipulate, i.e., labels, description, etc., but knowing that these fields contain not structured information, we preferred not to fully rely on them at the extraction moment.

[4]https://www.dhis2.org/downloads
[5]https://github.com/dhis2
[6]https://jira.dhis2.org/browse/DHIS2-1939

We queried the issues with the following constraints: Issue type = ‘Bugs’ or ‘Feature’, fixVersion = 2.27, Component like API* (i.e., issues related to changes in API component level).

Even though the extracted information was more detailed, it was not too well organized 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. Moreover, not all issues opened in JIRA were related to changes in API syntax.

- **Versioning systems**. We referred to Github commits’ history, to find information about changes in the API. We used ’Compare view’, a Github feature that gives the possibility to compare the repository across branches, commits, time, etc. But, this feature provides a comparison for the 250 most recent commits. We were interested only in API controller changes (at the level of API elements accessible for consumers), while the code repository consists of the whole API code (classes, methods, and functions not directly related to consumers). So we excluded this artifact from our analysis.

4.2.3 Which are the causes of the API changes?

From the four API artifacts in the study, we referred to the issue tracker to find the causes of the changes. Since only a small number of changes appeared in release notes, API documentation, and 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 comment we could find the issue ID of the issue opened at JIRA), the only possibility left was issue tracker.

We found more complete information in JIRA. The reasons were explained in the description of the issues or in the comments discussion. We checked every issue manually. As all project contributors (with or without technical background) can open issues at JIRA, the language used by them was not standard, thus making it difficult to create a unified set of causes. Anyway, during our work, we noted that there were some words, related to specific components of API, that were extensively used when these components were extended with new features or functionality. Thus, this phase can be further automated, but this is out of the scope of this paper. If we would rely on JIRA issue types, we would have a very coarse-grained classification of reasons: bug fixes and features. For bug fixes, the reason is clear enough: a bug somewhere in API causes an error, so API providers perform changes to fix it. For the features, the classification can be more detailed. Even though in JIRA, all these issues had the type "Feature", some of them were improvements or extensions of existing features. We used the usability taxonomy of Mosqueira-Rey et al. [64] and were able to fit every change in this classification, as in Table 2.2.

Table 2.2: 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**

Most of the issues in JIRA were related to the operability, robustness, and efficiency of the API.

As we can see, the changes extracted from JIRA are more than those identified from the controller comparison. This is because, in JIRA, issues are not always related to changes in the syntax of API. Besides this, when trying to match the changes from the API controller with the ones from JIRA, we noted inconsistencies between the two lists. Some of the changes in the API had their respective issues at JIRA, but opened as related to a different version or not related to API.

4.2.4 How are the API changes reflected in the usage logs?

We analyzed the Apache server logs of the DHIS2 system. The log format was well defined: "%h %t %r %>s %b", where:

- %h is the client's IP address;
- %t request time;
- %r request line, which contains the method used by the client, the resource requested, and the protocol used;
- %>s the status code that the server sends back to the client;
- %b the size of the object returned to the client.

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
```

The response given by the API:

```
{"dataElements":[
       {  "id":"HZYmbTohiAE",
          "displayName":"Anorexia "},
       {  "id":"Iedo09xfnkH",
          "displayName":"Anorexia(after)"} ] }
```

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 were 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 supports four last versions of the API, its consumers 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 whether consumers use it or not.

In version 2.27 a new feature was presented as follows (found in the release notes):

"Min-max data element values: API endpoint for getting/setting min-max values."[7]

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

In the 2.27 version, a new query parameter was added to analytics endpoint, `hideEmptyColumns` (found in API documentation). This parameter, when true, excludes from the response the columns which contain only null values. If we look at

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

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 to clients' requests.

On the other hand, having the set of changes that happen more often to APIs, and knowing how these changes appear in the logs, we can try to identify the need for these changes. We can monitor the consumers' behavior through the logs, and see where different change patterns can be applied. This can be very useful in automating the API evolution process.

5 Discussion

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

RQ1. We considered as ground truth the set of changes extracted from the API controller comparison. We want to note again, that accessing the API controller 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, a backward-compatible change as well. This can explain somehow the fact that most consumers had not been upgraded to the latest version of API. The creation of new elements (endpoints, parameters, etc.) is not a breaking change, hence 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 of them were documented in at least one of the other 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 though it is highly recommended for breaking changes to be documented [26,27,82], the addition of new elements is also seen by API producers as important to be documented. This way, API consumers can learn about these new elements and how to use them.

Venn diagram, in Figure 2.3, shows a comparison between API controller and API artifacts, in terms of the number of changes. We want to note that changes that were in release notes, and not in the API controller, were changes already implemented in previous versions or changes not related to API syntax. For example, two new endpoints `lockExceptions` and `email`, were already implemented in version 2.26, yet appeared as new ones in 2.27 release notes. The same happened with the issue tracker. Changes that were in JIRA and not in the API controller were changes not related to API syntax or changes already implemented, but miss-classified in JIRA as related to the 2.27 version.

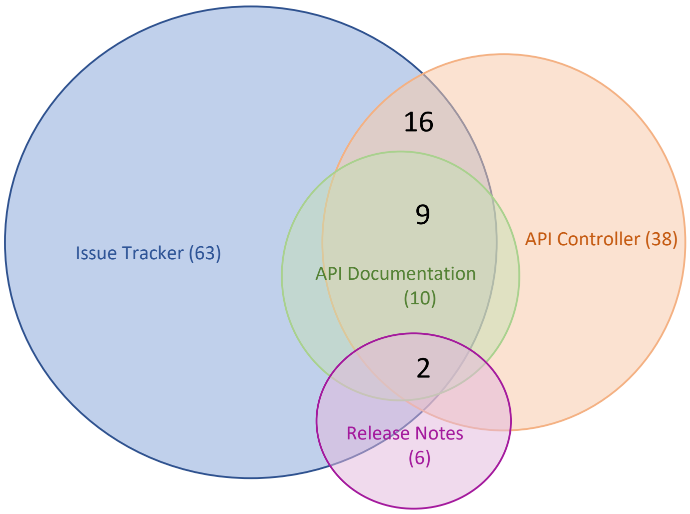


Fig. 2.3: API changes in API controller and API artifacts, Venn diagram.

The versioning system does not appear in the diagram, because as it is a code repository, changes extracted from it are the same as changes from the API controller.

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 the DHIS2 project, these four selected were the more complete and used ones.

During the analysis, we noted a lack of standard language used in describing changes in different artifacts. This complicated the matching process of the information from them. It would be very helpful if these artifacts would be more linked and synchronized with each other. Actually, there is a standard way of relating versioning system commits to issues as an issue tracker, but we saw that DHIS2 is not always using it. We saw that entries in the release notes of the DHIS2 system sometimes were followed by links addressing the API documentation section related to the change or to the issue opened at JIRA. Issues at JIRA sometimes included links to documentation also. If these standards would be the norm, it would be easier for API consumers to track the changes in these artifacts.

RQ3. In previous studies [15, 101], interviews were conducted with API producers, in order to get their intention for every change. Even though this would be 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. We saw that they tend to give such information in the issue tracker system (i.e., issue description, comments section). Their interactive and collaborative nature, mostly in discussion in the 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 the 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 the four last versions of the API, and few consumers did the upgrade to the new releases, thus benefiting from the newly introduced features. 10 changes out of 38 were adopted by consumers. These 10 changes were 3 new endpoints and 7 new parameters. Overall, more than 75% of the new changes preserve the backward compatibility of API. Until API providers support the previous versions, consumers will not feel the urge to upgrade, unless they really need the new features or the critical bugs fixed.

Threats to validity. In terms of work validity, the main threat to external validity is that we take into the study only one API. Moreover, we focus only on syntactical changes in API. Future work needs to be carried out to increase the data set in order to obtain more generalizable results.

The main threat to internal validity is related to the amount of manual work done. To alleviate this threat, we analyzed each change in different API artifacts.

6 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 of 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.

RQ1. We found 38 changes between version 2.26 and 2.27 of DHIS2. We classified them into six different types of change: new parameters, new endpoints, remove endpoints, new authority, change authority, and support request method. More than 75% of them were non-breaking changes (addition of new parameters or endpoints, respectively 19 and 10).

RQ2. In order to see to what extent were these changes documented, we analyzed four different API artifacts (namely release notes, issue tracker, documentation, and versioning system). Less than 50% of the changes were documented in at least one of the artifacts in the study.

RQ3. We used a well-known classification of API usability to classify changes based on the target usability aspect of the API they aim to change. Most of the changes were the addition of new features, thus enriching the API with new functionalities and increasing its operability. Changes were also performed in order to make the API more robust and to increase its performance, in terms of effort and time spent in consuming the API.

RQ4. Few consumers of DHIS2's API were upgraded to the latest release. As DHIS2 support the last four versions of API, they postpone the upgrade until they need the new features. From the adopted changes, we noted that consumers had difficulties in learning to use new endpoints, as the calls to them first resulted in 404 errors (client-side errors). On the other hand, the use of new parameters, in some cases decreased the size of the returned object. Actually, this is expected, as parameters are mostly used to filter the data.

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 in API syntax but also on changes in API behavior. By getting 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.

Chapter 3

Web API Change-Proneness Prediction

Abstract

Change-proneness of software artifacts has been mainly related to the design characteristics and their previous history of changes. While these two aspects are essential and contribute significantly to the prediction, they leave out a critical factor: how the artifacts are used. In the context of web APIs, consumers represent one of the main drivers of the change. Therefore, we propose a methodology for predicting the change-proneness of web API endpoint interfaces, taking into account not only design and change history but also their usage. Since the evolution of web APIs is and should be usage-driven, the way consumers use an API affects the future changes implemented by providers. Consequently, consumers' usage behavior contains essential information that contributes to identifying endpoints that are more prone to change. By considering the reasons behind changes, we introduce a set of metrics comprising design and usage aspects to be used as variables in prediction. We perform an initial evaluation using a real-world web API to demonstrate the approach's usefulness. We quantify the introduced metrics using web API documentation, code, and usage logs to build a classifier able to predict with 82% accuracy if an endpoint will change based on its design, history of changes, and usage characteristics.

1 Introduction

Web application programming interfaces (web APIs) undergo multiple iterations throughout their lifecycle, starting from the planning and design phases to their continuous releases. This iterative process is due to the fact that web API providers need to introduce various changes to their APIs for different reasons, including the evolution of the requirements set by the organizations that own the data or services, advancements in technology, competition-driven changes, and consumer demands. Providers must consider all these aspects when planning their new web API releases.

Indeed, consumers continuously provide explicit feedback (e.g., bug reports, and new feature requests) through various channels provided by the web API providers, including forums, tracking systems, emails, and feedback forms. Gathering, processing and effectively using this feedback, which originates from different sources, can be challenging. Providers have confirmed that they spend a significant time handling consumer requests and their bug reports [68]. Frequently, these requests are duplicated, lack relevance, contain errors, or are misclassified. Web API developers, under pressure to meet their key performance indicators (KPIs) (e.g., documentation completeness, release frequency), may prioritize other factors above the feedback provided by consumers. In fact, one of the key challenges identified by Lamothe et al. in their systematic review of API evolution literature is the need to exploit the feedback systems involved in API evolution [46]. Authors suggest focusing on these feedback systems (Lehman's 8th Law [49]), which now remain unexplored and present an impediment in API evolution. While web API providers understand the impact of changes on their consumers and the importance of considering consumers' needs when planning the changes [27, 53, 82], there is a clear opportunity to harness web API usage for creating more effective feedback loops that can help providers in improving their APIs [46].

This paper aims to show that consumers' implicit feedback (i.e., web API usage logs) plays an important role in the way web API endpoints (i.e., URL paths through which consumers can access the functionalities that web API implements) change through their lifecycle. This feedback that can be gathered and processed systematically captures all the interactions consumers have with endpoints. While explicit feedback shows consumers' subjective opinions, implicit feedback represents their objective behavior, i.e., how and when they actually use the web API [55]. By timely identifying endpoints more prone to change as per consumers' behavior, providers can prioritize these endpoints in the next releases, providing this way a web API more aligned with consumers' needs. With this approach, we aim to assist providers in better planning the evolution of their web APIs and effectively managing their resources (human, devices, time) during this process.

To implement our objective, we first consider a set of metrics commonly used in change-proneness prediction studies. These metrics are typically design-related or derived from the previous history of changes in the artifacts. We adapt these metrics to the context of web APIs and compute them from the documentation and code of the web API. Additionally, we define a set of usage-related metrics (e.g., number of consumers applications per endpoint, number of calls per endpoint) and compute them by processing usage logs data. Initially, we build two models separately, one for each approach (design model vs. usage model). Subsequently, we combine the two sets of metrics, to examine whether the addition of usage metrics improves the classical prediction models available in the state of the art.

In summary, the main objective of this research work is to show the relevance of usage in predicting the change-proneness of web APIs. To the best of our knowledge, the usage of web API has never been used in the change-proneness analysis. We approach this objective by answering three main research questions:

RQ1: Which metrics can be used for the change-proneness analysis of web API endpoints?

RQ1.1: Can we adapt existing approaches to fit into the context of web APIs?

RQ1.2: Which usage metrics can be used to predict web API change-proneness?

RQ2: To what extent can usage metrics improve the change-proneness prediction of web API endpoints?

RQ3: Which characteristics help to determine the web API endpoints that are more likely to change?

2 Background and Related Work

2.1 Core Concepts and Relationships

- **Objects** represent the data accessed through the web API. They define the structure and properties of the data and refer to the database tables or entities exposed.

- **Controllers** are software components that handle incoming requests and generate responses. They interact with the objects to retrieve or manipulate the data. Each controller can be associated with (depending on) one or more objects.
- **Endpoints** are URL paths that specify the location of the data to be accessed. They represent the entry point through which consumers communicate with the web API. Typically, one endpoint is associated with one controller class. Changes made to the controllers typically affect the interface of the endpoints [41]. In a few cases (e.g., improvement of the query to get the data), the change might not alter the interface, but it affects the behavior or the performance of the endpoint. In this work we use the concept of root endpoints, meaning the starting point or base URL (e.g., the root endpoint of `api/organisationUnits/ID/translations` and `api/organisationUnits/ID/dataSets` is `api/organisationUnits`).

Figure 3.1 exemplifies in a simplified way the main relationships between the three concepts. However, in practice, there can be additional components, such as models, services, and middleware.

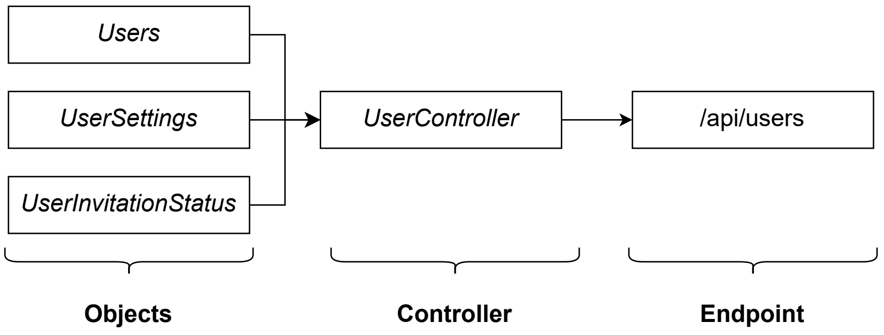


Fig. 3.1: Core Concepts and Relationships

2.2 Related Work

In this section, we discuss works focused on the change-proneness of software artifacts, web services, and APIs.

Change-proneness of software artifacts (e.g., classes, interfaces) has been broadly studied, as these changes can directly impact the behavior of the software using them.

Dealing with these changes usually requires a lot of time and high effort for the consumers. Therefore, identifying the artifacts that are more prone to changes becomes crucial for providers, not only to provide more robust artifacts in the future but also to allocate the available resources to implement the changes as soon as possible. In the following, we first present some concepts and their relationships. Then, we introduce representative works of change-proneness analysis in four different types of artifacts: classes, class interfaces, web services, and tables.

Malhotra and Khanna [59] performed a systematic review on software change prediction (SCP) to assess and evaluate the effectiveness of the existing works in predicting change-prone classes in software products. Typically, SCP models use structural, network, evolution, word vector, and developer metrics. Remarkably, none of the mentioned works considered usage-based metrics. In an earlier study [57], the same authors studied the relationship between object-oriented metrics and class change-proneness. They applied their approach to three open source projects, and found the most significant indicators of change-proneness (e.g., Response For a Class RFC) as the methods implemented within a class and the number of methods accessible due to inheritance).

Arvanitou et al. [9] assessed the change-proneness of classes based on the history of their changes, and the structure of their source code. They introduced a Change Proneness Measure (CPM) based on the internal and external probability a class may change and compared its accuracy with the accuracy of using other metrics (e.g., coupling metrics, only history of changes). They noted that both history of changes and structural metrics are needed for an accurate assessment. Abbas et al. [1] focused more on the best techniques for change-prone prediction using object-oriented metrics (e.g., number of lines of code, weighted methods per class), and concluded that machine-learning methods are beneficial in change prediction.

Romano and Pinzger [77] investigated the correlation between source code metrics and change-proneness of Java interfaces (not the implementation of the classes). They considered metrics defined by Chidamber and Kemerer [19] (e.g., number of methods in a class, depth of inheritance tree), and also complexity metrics (e.g., number of classes that invoke the interface, number of classes that implement the interface). They showed that cohesion metrics, different from Chidamber and Kemerer metrics, had a strong correlation with Java interface change-proneness, reinforcing the authors' claim that classes and interfaces change-proneness should be analyzed separately as they are influenced by different indicators.

Vassiliadis et al. [94] studied the relation between table properties (e.g., the number of attributes, the time of birth of a table) and evolution-related properties (e.g., the possibility of deletion, the number of updates). Different from us, they do not consider how the tables are being used and also consider a few changes (only additions and deletions of attributes, data type, and key changes).

In summary, our analysis of the related work reveals a notable gap in change-proneness prediction research: none of the studies explored usage as an indicator for change-proneness. This highlights the need for future investigations to incorporate usage metrics in order to enhance the accuracy of change prediction models.

3 Framework

In this section, we respond to RQ1. We start by adopting the web API domain metrics already used in the change-proneness analysis of software artifacts [9], [8], [94]. As mentioned, these analyses take into account mostly design characteristics (the size of the classes, the dependencies with other classes, etc.). Then, we introduce other metrics that characterize and target specifically the web API usage.

Table 3.1 introduces the set of metrics used as features in the model for change-proneness prediction. We make use of the change-proneness measure (feature 1, CPM) introduced by Arvanitou et al. [9]. CPM is calculated at the class level, and combines information about the class change history (i.e., the changes that have been implemented to the class before) and class dependencies to other artifacts (i.e., the part of the interface of a class that is used by other classes), as the changes can be propagated from other artifacts. As the authors proved, this combined metric was the most correlated with the class change-proneness, because both aspects it covers (i.e., history of changes and class dependencies) are needed to accurately predict class change-proneness.

Arvanitou et al. [9] define the CPM of class C (in our case a controller class that corresponds to an endpoint) as the joint probability of the probability of the class itself to change, P(C), and its external probability to change, $P(C : external_{D_i})$, where i=1 to N is the number of D classes that C depends upon. Instead, we do not include P(C) in the CPM calculation (3.1), but count it as a feature itself (feature 2), to have separately the internal and external probability to change. Moreover, unlike [9] which uses the percentage of commits, we calculate P(C) as the percentage of lines changed in the previous release, as it offers a more consistent measure of the internal probability of a class to change, regardless of the development team's approach, be it agile or traditional.

$$CPM(C) = JointProbability\{P(C : external_{D_i})\} \tag{3.1}$$

Table 3.1: Design and usage metrics

Nr.	Metrics	Type
1	change-proneness measure	CPM
2	history of changes	evolution
3	number of required attributes per endpoint	design
4	number of attributes per endpoint	design
5	number of required parameters per endpoint	design
6	number of parameters per endpoint	design
7	average number of calls per day per endpoint	usage
8	number of consumers applications per endpoint	usage
9	number of cases per endpoint	usage
10	number of calls per endpoint	usage
11	average number of attributes used in queries	usage
12	maximum number of attributes used in queries	usage
13	average number of parameters used in queries	usage
14	maximum number of parameters used in queries	usage
15	average query length	usage
16	average object size returned per endpoint	usage
17	maximum object size returned per endpoint	usage
18	standard deviation of object size returned per endpoint	usage
19	coefficient of variation of object size returned per endpoint	usage
20	distinct endpoints called directly after the endpoint	usage
21	current changes	evolution

For each class D_i that C depends on, we calculate $P(C : external_{D_i}) = P(C|D_i) * P(D_i)$, where $P(C|D_i)$ (called Ripple Effect Measurement) is the propagation factor between class D_i and C, and $P(D_i)$ is the probability that class D_i changes. We adopted $P(C|D_i)$ calculation from [8].

Besides CPM, we include in the analysis four other design features that describe the number of attributes and parameters, as web APIs design characteristics (features 3-6). We adopted these features from [94], where the authors consider tables' attributes to assess the possibility of tables to change. Thus, we consider optional and required attributes (used to filter the attributes of the data instances), and optional and required parameters (used to filter the data instances returned).

Regarding the usage of web API endpoints, we define metrics that characterize consumers' interaction with endpoints in several aspects. Features 7 to 10 describe the frequency of consumers' interaction with the web API endpoints (e.g., calls per day per endpoint, cases per endpoint). Features 11 to 15 describe the details of the query part in the web API requests (i.e., how consumers use the available parameters and attributes in their requests). Features 16 to 19 describe the response returned by the web API (e.g., the size of the object returned). Feature 20 describes how consumers use one endpoint in relation to the other endpoints. It measures the frequency of distinct endpoints that are directly called after one endpoint in the usage logs. Lastly, feature 21 acts as the class variable and tracks the changes of the last release.

Figure 3.2 describes the data analysis steps. The upper part (red arrows) depicts the preparation steps, which mainly consist of metrics quantification (Section 4.2). The bottom part depicts the prediction model creation. We applied machine learning methods, as they have been proven to give accurate results in change-proneness prediction [1]. Moreover, we opted to create a classification model, as we had predefined classes with discrete labels, and we were interested in finding if an endpoint was prone to change or not, more than predicting the exact probability of an endpoint to change (as a regression model would provide). To obtain highly interpretable results, we employed a decision tree algorithm.

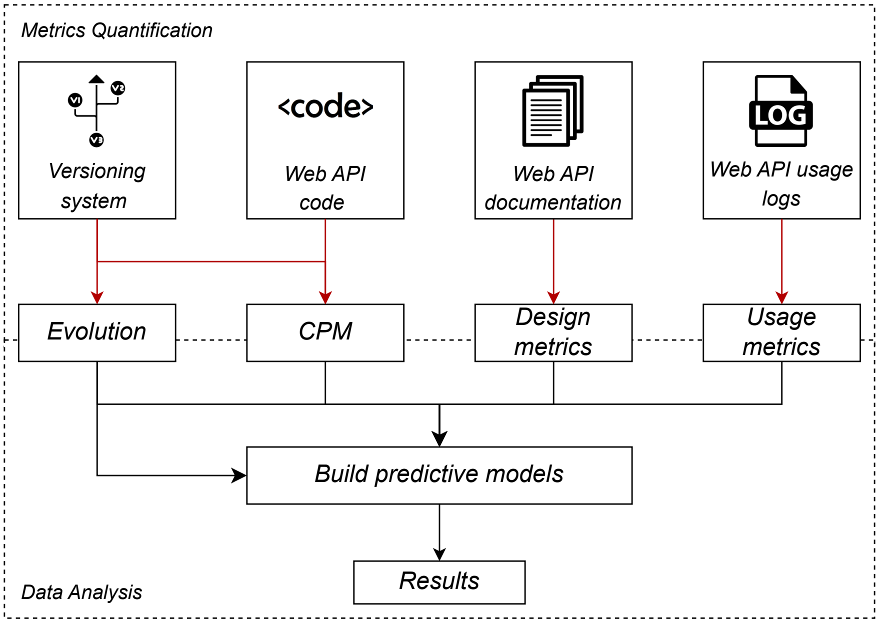


Fig. 3.2: Framework

4 Evaluation

4.1 Use Case

To evaluate our framework and the significance of the introduced metrics in prediction, we analyze the District Health Information Software 2 (DHIS2) web API.[1] Usage logs are from February to September 2019, and the logs together contain approximately 2.5 million requests of version 2.30 of the DHIS2 web API. The reason to focus on usage logs from earlier versions of the web API is that we can better observe the impact of the usage on several later releases. DHIS2 releases a new version of the web API twice per year.[2] They have currently released version 2.40.

4.2 Metrics Quantification

To quantify the usage metrics (features 7-20), we refer to the usage logs. We first extract the root endpoints for which we have requests in the log file. Overall, there are 117 different root endpoints. For each of them, we calculate all introduced usage metrics.

To quantify CPM, we first extract the dependencies of controllers, and then the history of changes. To extract the dependencies of controllers, we use the Dependency Finder tool,[3] giving as input the Java bytecode of the web API. For each controller, we obtain the objects classes the controller depends upon, and the methods and variables the controller uses from them. To extract the history of changes of objects and controllers, we compare their Java files of previous pairs of versions (until 2.30), and calculate the average percentage of lines each of them has changed.

To quantify the rest of the design metrics (features 3-6) for the DHIS2 web API, we first refer to the DHIS2 OpenAPI Specification project.[4] For the root endpoints that are not specified there, we consult web API documentation. After quantifying these four metrics, we observe their values. For the number of parameters (required and optional), DHIS2 web API has the same value for all the endpoints (the number of parameters was at the web API level, not the endpoint level). Thus, we exclude them from the analysis. We want to note that providers might have more systematic data (e.g., dependencies between controllers' classes can be part of project documentation, not made public), and might quantify the metrics using other sources. We

[1]https://docs.dhis2.org/en/develop/using-the-api/dhis-core-version-master/introduction.html

[2]https://dhis2.org/downloads/archive

[3]https://sourceforge.net/projects/depfind/files

[4]https://github.com/dhis2/dhis2-api-specification

demonstrate how we compute the metrics using the available sources, so we can validate our approach.

To identify the endpoints that changed (i.e., to quantify the class variable, feature 21), we refer to the controller code of the DHIS2 web API. For each controller, we detect if it changed from version 2.30 to 2.31.

Of the 117 different root endpoints, the controller of 92 of them had not experienced any changes, while the controller for the remaining 25 endpoints had changed from version 2.30 to version 2.31. As the dataset was significantly imbalanced (the minority class was 21.4% of the dataset), we had to balance it to have accurate results [58]. As we did not have too many endpoints, we opted for oversampling (adding entries in the minority class), instead of under-sampling (removing excessive entries from the majority class). We performed SMOTE on the dataset (only on the training set), as one of the most efficient balancing techniques [78], and then built a classifier, a decision tree, using WEKA libraries implementation in JAVA. To avoid over-fitting and over-optimistic results, we performed 10-fold cross-validation.

4.3 Data Analysis

In the following of this section, we respond to RQ2 and RQ3. In order to get a clear view of the role that usage has on change-proneness, and whether providers are taking it into account or should focus more on it, we performed three different analyses.We built three classifiers, one using only design and history of changes metrics (i.e., design model), another only with usage-related metrics (i.e., usage model), and then a third one with all metrics combined (i.e., combined model). The combined model was the best one, with 82% accuracy (balanced accuracy 82%). The design model had a 77% accuracy (balanced accuracy 74%), while the usage model performed an accuracy of 63% (balanced accuracy 49%). Even though we applied SMOTE to mitigate the class imbalance issue, we give also models' balanced accuracy, to show that in our case accuracy is not a biased measure. The first insight we can get from these results is that the combination of design and usage metrics enables a more accurate prediction of future changes in an endpoint, emphasizing their complementary role and highlighting the limitation of relying on a single set of metrics. Furthermore, the design model performing better than the usage model suggests that the changes implemented by providers are more design-related. Consequently, providers may not adequately consider the usage of their web APIs, resulting in a discrepancy between consumer behavior and the types of changes implemented.

To better compare the design and usage model, we use an UpSet graph [52] to visualize their positive results. In Figure 3.3, we observe that three true positives

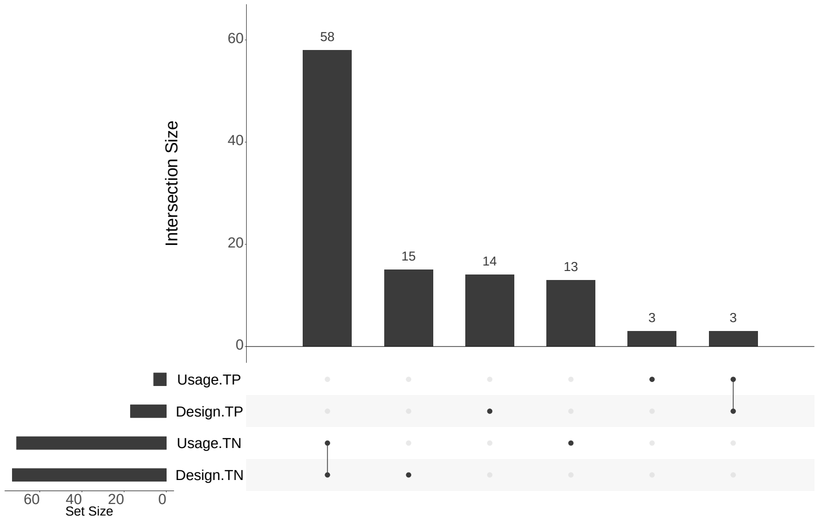


Fig. 3.3: True Positives and True Negatives

(fifth bar) and thirteen true negatives (fourth bar) identified by the usage model are not detected by the design model. This finding suggests that the usage model makes distinct contributions to the classification process.

In order to get more insights about the false negative predictions (i.e., changed endpoints that the combined model was not able to predict), we analyzed in detail the respective endpoints. The five false negative predictions were changes made for improving the consistency and standardization of the web API. For instance, the endpoint `userSettings` (which had changed but was predicted as stable), was modified so that it would be consistent with some other endpoints in the web API ("DHIS2-4982[5] Align userSettings with systemSettings API: The `api/systemSettings` API supports "key" params so that you can reduce the response payload by requesting specific settings. The `api/userSettings` API should support this as well."). Additionally, the pagination parameter was added to endpoints `completeDataSetRegistrations` and `programDataElements`. Typically, providers add pagination parameters to endpoints to help consumers manage large amounts of data returned by the web API. In our case, providers have added pagination to make the endpoints more consistent with the other endpoints in the web API.

Driven by our observation that several issues in JIRA were resolved in later versions after their reporting, and supported by existing literature research [68], we examined the later versions of the DHIS2 web API, from version 2.32 to 2.34. This al-

[5]https://dhis2.atlassian.net/browse/DHIS2-4982

lows us to also better understand and interpret the false positive predictions (i.e., endpoints that were predicted changed but remained unchanged). All of the endpoints that our model predicted as likely to change had indeed undergone changes in the subsequent three releases. To validate the aforementioned interpretation, we built an additional classifier with a specific modification. In this modified classifier, we quantified the class variable 'change' based on endpoints that had undergone changes in more than one version. If an endpoint had experienced at least one change from the release of version 2.31 to version 2.34, the value of the variable was assigned as 'changed'. Conversely, if an endpoint remained unchanged during this time frame, the value was designated as 'stable'. The modified model achieved an accuracy of 89% in predicting endpoint change-proneness. The fact that endpoints that eventually changed in the subsequent releases had previously exhibited usage behavior from consumers demonstrates the promising potential of our approach. The addition of usage metrics in the model, not only improves its accuracy but enables providers to plan timely the needed changes.

To address RQ3, we analyzed the tree branches for each model. In the design model, CPM was the most important feature. Controllers with a high CPM were more prone to change. In fact, controllers dependent on multiple objects (fat controllers) are considered a bad design practice [6]. Our results reinforce this statement. As for the usage model, the number of calls per day, object size, and the number of parameters were among the most important features. Endpoints with a high number of calls per day, large object sizes, and few parameters were the ones more prone to change. In the combined model, even though the design metrics remained the primary determinants of the decision-making process, usage metrics appeared as contributing factors within the branches of the decision tree. Interestingly, the presence of usage metrics increased in the modified model.

5 Threats to Validity

One potential threat to the validity of our study is the set of metrics used. While we included metrics that cover most of the design and usage characteristics, it is important to acknowledge that there may be other metrics not considered in our analysis that could impact the prediction. Specifically, metrics related to business requirements were not included, as they are highly domain-specific and cannot be adequately described using generic metrics. Our focus in this paper is primarily on understanding the impact of usage and design on the change-proneness of web APIs. To ensure a comprehensive set of metrics, we referred to prior research on change-proneness prediction and adapted relevant design and usage metrics into the web API context.

Another threat to the validity concerns the dataset used for training and testing. We acknowledge that the ideal approach would have involved training the model on one release and evaluating it on a different one. However, due to the availability of log data only from version 2.30, we were constrained to use data from the same release for training and testing. Nevertheless, to avoid over-fitting and over-optimistic results, we performed 10-fold cross-validation, making sure to properly separate the testing set from the training one, so as not to have biased results.

6 Conclusion

Change-proneness has traditionally been tied to the design and structure of artifacts, where poor design practices and specific characteristics can lead to unstable artifacts and frequent changes. While this holds true for software artifacts like classes and methods, which are used within software systems and exhibit interdependencies, it is important to also consider a crucial factor when analyzing web APIs: consumer usage. The behavior of API consumers plays a significant role in the evolution of web APIs, and its impact should not be overlooked when building predictive models.

While models constructed based only on design metrics demonstrated better performance than models using usage metrics, it is noteworthy that changes predicted by the usage model were not captured by the design model. This indicates that usage has its unique contributions to the classification task. Thus, to explore its full potential in change prediction, we extended the analysis to include changes spanning multiple releases.

Our preliminary findings indicate that incorporating usage metrics into the prediction model not only improves accuracy but also enables early prediction of changes that providers implement at later stages. By considering both design and usage metrics, API providers can make proactive decisions, ensuring that their changes align with user expectations.

Chapter 4

Improving Web API Usage Logging

The paper has been published in the *In Proceedings of the International Conference of Research Challenges in Information Science (2021).*
DOI: http://dx.doi.org/10.1007/978-3-030-75018-3_45

Abstract

A Web API (WAPI) is a type of API whose interaction with its consumers is done through the Internet. While being accessed through the Internet can be challenging, mostly when WAPIs evolve, it gives providers the possibility to monitor their usage. Currently, WAPI usage is mostly logged for traffic monitoring and troubleshooting. Even though they contain invaluable information regarding consumers' behavior, they are not sufficiently used by providers. In this paper, we first consider two phases of the application development lifecycle, and based on them we distinguish two different types of usage logs, namely development logs and production logs. For each of them, we show the potential analyses (e.g., WAPI usability evaluation) that can be performed, and the main impediments that may be caused by the unsuitable log format. We conduct a case study using logs of the same WAPI from different deployments and different formats, to demonstrate the occurrence of these impediments and at the same time the importance of a proper log format. Next, based on the case study results, we present the main quality issues of WAPI logs and explain their impact on data analyses. For each of them, we give some practical suggestions on how to deal with them, as well as mitigating their root cause.

1 Introduction

An increasing number of organizations and institutions are exposing their data and services by means of Application Programming Interfaces (APIs). Different from traditional APIs (i.e., statically linked APIs), which are accessed locally by consumers, web APIs (WAPIs) are exposed, and thus accessed, through the network, using standard web protocols [88]. As the interaction between WAPIs and their consumers is done typically through the Internet, both parts end up loosely connected.

This loosely coupled connection becomes eventually challenging, mostly during WAPI evolution, when as a boomerang effect, consumers end up strongly tight to WAPIs [27]. If providers release a new version and decide to discontinue the former ones, consumers are obliged to upgrade their applications to the new version and adapt them to the changes. Consequently, WAPIs end up driving the evolution of their consumers' application [26], [25]. Knowing the considerable impact WAPIs have on their consumers, providers would benefit from consumers' feedback to understand their needs and problems when using the WAPI [65].

Currently, API providers face a lot of difficulties in collecting and analyzing consumers' feedback from several sources (often informal ones), e.g., bug reports, issue tracking systems, online forums, and discussions. [102]. Furthermore, feedback collection and analysis turn out to be expensive in terms of time, thus difficult to scale. Actually, in the WAPI case, this feedback can be gathered in a more centralized way.

While being accessed through the network poses some challenges for consumers, it enables providers to monitor the usage of their WAPIs, by logging every request that consumers make to them (see Fig. 4.1). WAPI usage logs, besides coming from a trustworthy source of information, can be gathered in a straightforward, inexpensive way, and completely transparent to WAPI consumers.

Currently, WAPI usage logs are mostly being used to feed monitoring tools. These tools typically provide automatic alerts when WAPI endpoints fail, and reporting dashboards that visualize several performance metrics [23]. Since these logs are not designed to be further analyzed with regard to consumers' behavior, they may lack some critical information, like identifiers for consumers' applications, etc. Moreover, they are complex (e.g., unstructured, high-volume data) and somewhat noisy. Applications' design and the way their end users use them can veil interesting and important WAPI usage patterns. Therefore, understanding consumers' behavior and inferring their needs from WAPI usage logs becomes essential for providing high-quality WAPIs, tailored to the real needs of their consumers.

This paper is building upon our previous work [42], where we measured the

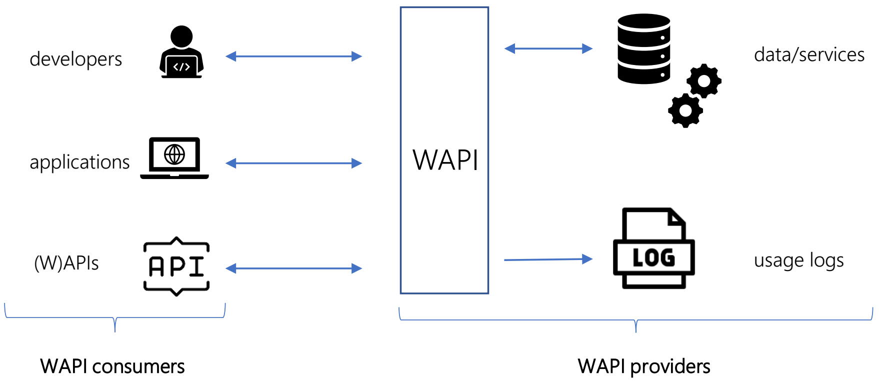


Fig. 4.1: The interaction between consumers, providers, and WAPIs

usability of WAPIs by analyzing their usage logs generated during the development phase of consumers' applications. Based on the challenges faced while working with WAPI usage logs, and the surprising lack of attention this topic (i.e., WAPI usage logs analysis) had gained, we saw it convenient to summarize our experience and research in the field, into a set of practical suggestions to enhance the logging of WAPI usage for more specialized analyses.

To this end, the contribution of this work is fourfold. We focus first (i) on showing the potential and then (ii) on the main impediments of proactively using WAPI usage logs regarding consumers' behavior. We then conduct a case study using logs of the same WAPI from different deployments, using different log formats, (iii) to show the importance of logs' structure and content in preparing the data for further analyses. Next, based on the case study findings, the analysis requirements, WAPI logs structure, and also after reviewing the relevant literature, (iv) we classify and define the main issues and obstacles that hinder the application of various analyses on the logs. For each of the issues, we describe the impact their occurrence may have on the analysis result, and propose mitigation actions.

The remainder of this paper is organized as follows. In Section 2, we report on the current use of WAPI usage logs and practices to pre-process and deal with their quality issues. In Section 3, we give the main motivation behind our work. We introduce the WAPI logs structure and content, and based on them we propose some purpose-specific analyses that can be applied to these logs. In Section 4, we introduce two pre-processing challenges, whose accuracy may be affected by the format of the logs. In Section 5, we introduce our case study and the experiments we performed. In Section 6, we provide the set of WAPI quality issues inferred from the case study and discuss their impact and mitigation. In Section 7, we conclude the paper and present some ideas for future work.

2 Related Work

Few works build their analysis on WAPI usage logs comprised of the URL requests made to the WAPI [87], [42], [56]. Thus, not only these logs' potential is still unrevealed, but even quality issues related to them or their pre-processing and preparation have not gained the deserved attention.

For instance, Suter and Wittern [87] used the usage logs to infer the WAPI description (the endpoint structure and parameters) from them. They reported that the results of their methods were impeded by the incomplete and noisy nature of these log data. In our previous paper [42], we proposed an approach to measure the usability of WAPIs by analyzing the usage logs generated during the development phase of consumers' applications. We described the pre-processing steps of WAPI usage logs in general, and then demonstrated how we dealt with specific obstacles in the log data from the case study (e.g., data structuring, and generalization). Macvean et al. [56] analyzed WAPI usage logs from Google API Explorer, and generated from them several structural factors (e.g., number of parameters, number of methods) to study their usability. Even though they showed the potential of analyzing these logs, they did not tackle the quality issues related to them or challenges during their preparation.

On the other hand, web log preparation and pre-processing are widely studied, as part of web log analysis, and extensively applied regarding usage and usability of software and web applications [31]. We mention below some of these works, as some WAPI logs quality issues are similar to the web logs ones.

One of the most discussed issues of log pre-processing is session identification. A session represents the interaction of a user with a website within a time frame (usually expires after a certain amount of time of inactivity). There exist several heuristics for reconstructing sessions, mostly coming from web mining applications [85], [84], [89], [40], [11], [83]. Spiliopoulou et al. [83] applied different heuristics (total session duration, page-stay duration, etc.) to reconstruct the sessions from the server log data. They evaluated the performance of these heuristics to the server log of a university site by comparing the reconstructed sessions with the real ones. Their experiments showed that there is no one best heuristic for all cases, and it depends on the site's structure and traffic. Kapusta et al. [40] analyzed the logs from a commercial bank portal to identify the users' sessions. They applied different time window thresholds and heuristics, and based on the usefulness of the rules extracted in each case, they evaluated the best threshold. Tanasa and Trousse [89] described in detail all the steps of log pre-processing (i.e. data fusion, data cleaning, data structuring, and data generalization), indicating the most challenging issues and how to overcome them. They pointed out the importance of pre-processing in data analysis effectiveness, and among others, agreed on the need for better log systems.

The above-mentioned works focused on general web usage logs, thus the concerns raised were related to the analyses applied to them (e.g., to identify users' navigation behavior in order to predict their next actions, to evaluate software design usability, to monitor the traffic for performance reporting). While some of the issues of working with general web logs are similar to WAPI usage logs, the latter poses some added challenges, related to the requirements of the specific analyses that can be applied to them, as well as the WAPI design.

Bose et al. [14] focused their work on the requirements of process mining (an analyzing technique that can be applied to log data) and log data quality issues that may affect its results. They presented 4 process characteristic issues and 27 event log quality issues that hinder the applicability of several process mining techniques and affect the results' quality, but did not provide solutions on how to address those issues. Along similar lines, Suriadi et al. [86] described a set of data quality issues, frequently found in process mining event logs. Based on their experience in performing process mining analyses, they introduced 11 event log imperfection patterns, which can be used in several domains. While both of the works are too specific for process mining requirements, they refer to event logs in general in terms of the domain. Thus, some of the issues introduced cannot be applied in the WAPI domain (e.g., issues related to manual data entry).

In this paper, driven by the lack of attention WAPI usage logs analysis has gained, we show several potential analyses that can be applied to them, mostly based on their nature and content. We discuss the posed challenges in analyzing these logs and eliciting the needed information, followed by recommendations on how to better log the WAPI usage, which until now, remains quite unexplored.

3 The potential of WAPI usage logs

Providers typically log their WAPIs' usage by recording all requests done against the WAPIs. Every time a consumer's application issues a request to a WAPI, a log entry is generated and stored in the usage log file. The information that is logged for each request and its format may vary due to different logging system setting parameters and also providers' decisions about logs design. For example, using the

Apache Custom Log Format[1] (a flexible and customizable format), when an application makes a request like `https://maps.googleapis.com/maps/api/distancematrix/json?origins=MNAC&destinations=MACBA&mode=driving&key=API_IDENTIFIER` (in the form of a URL) to a WAPI endpoint of the Google Maps Platform, the following information could be logged by providers (based on the logging system configuration): the IP address of the application's user, the time when the consumer made the request, the request body that contains the request method (GET), path (`maps.googleapis.com/maps/api/distancematrix/json`) and query (`origins=MNAC&destinations=MACBA&mode=driving&key=API_IDENTIFIER`), the protocol (HTTP/1.1), the time needed to respond to the request, the status code (e.g., 200 if the request was successful), the size of the object returned, the address of the page that initiated the request, information about the operating system or browser used, and other fields comprising different aspects of consumers' applications and their users.

We can think of usage logs as traces that consumers leave when using the WAPI. If the information in these traces is analyzed in the proper way, it can reveal useful knowledge. They show which endpoints the consumers have accessed, in which order, with which frequency, and with which parameters. As applications are the actual WAPI consumers, we should consider the different ways they consume WAPIs over their own lifecycles. Basically, applications interact with the WAPIs during design time and runtime, over both of which manifest different aspects of their behavior. Following on from this, we distinguish two types of logs: (i) development logs, and (ii) production logs (Figure 4.2).

Development logs are generated at design time, while developers build and test their applications. During this phase, they make the first integration of the WAPI into their applications or implement newly developed features. Thus, these logs show their attempts in using the WAPI, the endpoints they struggle more with, specific mistakes they do while using and learning the WAPI, etc. [42], [56]. By analyzing these logs, providers may evaluate the usability of their WAPI from the consumers' perspective. For instance, they may decide to change the name of elements (endpoints, parameters) consumers have difficulty learning or memorizing, improve the documentation for endpoints that seem not clear to consumers, detail the error messages when they detect that consumers are repeating continuously the same errors without understanding how to fix it, etc.

On the other hand, production logs are generated during application runtime,

[1]`http://httpd.apache.org/docs/current/mod/mod_log_config.html`

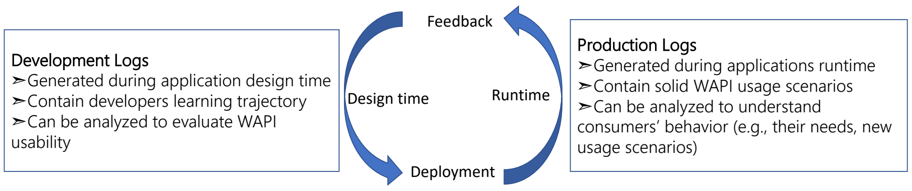

Development Logs
➢Generated during application design time
➢Contain developers learning trajectory
➢Can be analyzed to evaluate WAPI usability
Feedback
Design time
Runtime
Deployment
Production Logs
➢Generated during applications runtime
➢Contain solid WAPI usage scenarios
➢Can be analyzed to understand consumers' behavior (e.g., their needs, new usage scenarios)

Fig. 4.2: The development lifecycle of consumers' application

while applications are being used by end users. Since the applications are released for public use, it is assumed that they are quite steady, without erroneous WAPIs requests. WAPI requests are predetermined by the implemented functionalities of the applications, different from the development phase, where developers may freely try different requests, several times, and pose any query. Indeed, production logs contain real and solid WAPI usage scenarios, the right order in which developers make the requests to WAPI to achieve specific goals, different workarounds created to accomplish tasks for which there are no WAPI endpoints developed, or the actual frequency of certain requests or sequences of requests, that show the real consumption of WAPIs. By analyzing these logs, providers may identify consumers' needs for new features, and implement the corresponding endpoints. These logs may reveal new usage scenarios providers may have not thought about before, instructing them in including these scenarios in the documentation. Besides these, providers may identify ways of improving the WAPI based on how consumers use it, merging endpoints that are always called together for a specific purpose, or creating new endpoints, derivative from the ones that are always called with specific values for some parameters.

Even though both types of logs provide useful information about WAPI consumption and perception from consumers, preparing and analyzing them is arduous. First, it is not always trivial to distinguish these logs from each other, as they often are stored together in the same files. Secondly, consumers' applications design and the way users interact with them will be manifested in the production logs, obfuscating the inference of the real WAPI usage patterns. Providers should identify the patterns that represent real usage scenarios, from the ones deriving from applications design and user flow. Finally, as providers store these logs typically for traffic monitoring, they do not consider the requirements that specific analyses may have. Thus, unawarely, they may neglect the importance of the log format, and even leave out crucial information for consumers' identification, adversely affecting not only the analysis results but also the logs' pre-processing.

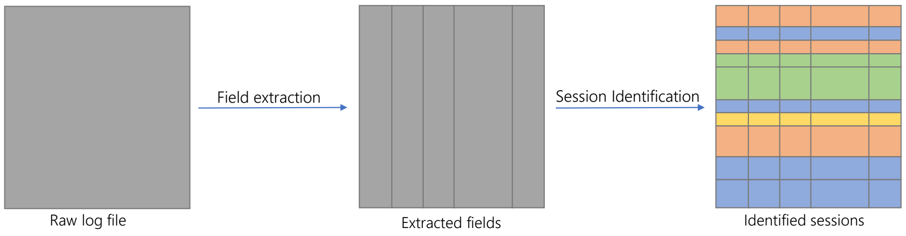


Fig. 4.3: Field extraction and session identification

4 How does the logs format affect the pre-processing?

The pre-processing phase is typically counted as the most difficult and time-consuming part of log analysis [85]. It basically consists of four main steps: (i) data fusion, consisting in gathering and merging log files from different sources, (ii) data cleaning, consisting in removing irrelevant data and completing missing values, (iii) data structuring, consisting in segmenting the log file in users' sessions, and (iv) data generalization, consisting in generalizing the dynamic part (i.e., parameter values) of requests [89], [42]. In this section, we will cover two challenges from WAPI usage logs pre-processing, namely field extraction from the data cleaning phase, and session identification from data structuring, as the two challenges of pre-processing that, are directly affected by the log format and the way the usage is being logged (Figure 4.3).

Field extraction. Usage logs are stored in text files. Each log entry contains several fields, each containing specific information. Field extraction consists exactly of the separation of the log entry in several fields. It is typically performed right before data cleaning so that log entries can be filtered based on the value of their specific fields (e.g., the request method, and request body). For example, the following log entry should be transformed from a single string into the set of fields it contains `127.0.0.1 - - [24/Jun/2019:20:22:26 +0000] GET /api/29/system/info HTTP/1.0 200 891 https://.../dhis-web-dashboard/index.html Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3770.100 Safari/537.36`, extracted fields:

```
Client IP  : 127.0.0.1
```

```
Timestamp  : 24/Jun/2019:20:22:26 +0000
Request    : GET /api/29/system/info HTTP/1.0
Status code: 200
Object Size: 891
Referer    : https://.../dhis-web-dashboard/index.html
User-Agent : Mozilla/5.0 ... Safari/537.36
```

There are several ways to parse log file information, including regular expressions, predefined parsers, and custom grok parsers (pattern matching syntax used by ElasticSearch). Providers should decide on a log format that can be easily parsed, in order to enable simple querying of fields value.

Session identification. This challenge (often called *sessioning*), refers to grouping together into the same session, all the log entries (i.e., requests) coming from each user during the time frame of a visit, trying not to leave out any log entry, as well as not assigning wrong ones. It is one of the main issues with WAPI usage logs. Most of the WAPIs are stateless, meaning that the server does not store the state, thus no sessions are generated. The lack of session identifiers may seriously impede the applicability of several analyses. For instance, one of the process mining requirements is for event logs to have case identifiers, which assign each log entry to a specific case [92]. Session identifiers must be inferred by combining other available information in the logs. Sessioning heuristic is a method for constructing sessions based on assumptions about users' behavior or the site/application characteristics. Two of the most applied methods are time-based heuristics and navigation-based heuristics [11]. As both of them are built under the hypothesis of an already launched and ready-to-be-used application, these heuristics apply only to the production logs.

- *Time-based heuristics* construct the sessions based on either: (i) the duration of a user's entire visit to the application, which should not surpass a maximum threshold δ, typically taken 30 minutes [11], or (ii) the time a user spend on one page of the application (i.e., page-stay heuristic), which should not surpass a maximum threshold θ, defined based on pages average contents and application nature.

- *Navigation-based heuristics* construct the sessions based on the assumption of how the applications' pages are related. The rationale behind this is that users' navigational flow in the application has been predetermined since its implementation. For native (or desktop) applications this flow is fixed. For web ones, which are accessed through a browser, users rarely type themselves the URL of a page but rather follow the hyperlinks and the navigation bar. In the usage logs, the information about the page initiating the actual request is contained under the referer field. This field can have a null value ("-") when the users type the request directly in the browser, or when an application is first opened.

5 Case study

We perform field extraction and session identification in order to demonstrate the importance of specific fields of the log format, and the impact their lack may cause to both of these challenges. We conduct a case study using logs of the District Health Information Software 2 (DHIS2) WAPI. DHIS2 is an open-source, web-based health management information system platform used worldwide by various institutions and NGOs for data entry, data quality checks, and reporting. It has an open REST WAPI, used by more than 60 native applications. External software can make use of the open API, by connecting directly to it or through an interoperability layer.

DHIS2 is instantiated as World Health Organization (WHO) Integrated Data Platform[2] (WIDP), and is used by several WHO departments for routine disease surveillance and country reporting. For the analysis, we use the production logs from WIDP, and from Médecins Sans Fontières (MSF), another DHIS2 instance used for field data collection and as a central repository for medical data.

Both of these instances use the same DHIS2 WAPI and the same set of applications accessing it. But, being deployed and used independently, the logs coming from them have different formats, providing us with different information that we can use to structure and prepare the logs for further analyses (Table 4.1).

Table 4.1: Log formats of the two DHIS2's deployments under study

Deployment	Client IP address	Timestamp granularity	Duration	Request	Status Code	Object Size	Referer	User Agent
MSF	✗	Second	✓	✓	✓	✓	✓	✓
WIDP	✓	Millisecond	✓	✓	✓	✓	✗	✓

1. **Field extraction.** We perform field extraction by using regular expressions in JAVA. The request body, referer, and user-agent are the parts that generate more errors while parsing, as they may include spaces and special characters, sometimes used to separate the fields. We show the example of the user-agents values in WIDP

[2] `http://mss4ntd.essi.upc.edu/wiki/index.php?title=WHO_Integrated_Data_Platform_(WIDP)`

log data, which typically have in their body comma, semicolon, and spaces: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.122 Safari/537.36. We have to perform some extra manual work to handle the errors, like splitting these fields into several parts and then joining them, without cluttering parts of different fields.

2. **Session identification.** We apply the page-stay heuristic and the navigation one combined with time constraints. We perform the experiments on log data from MSF since the log format contains information about the referer (Table 4.1), required in the navigation-based method. Since we do not (and cannot) have the data with the real and correct session identifiers to evaluate the performance of the resulting techniques, we assess the correctness of the constructed sessions based on different statistics. The results of the analysis are shown in Table 4.2.

   The WAPI log file from MSF has requests from different applications installed in the platform and used by different users. Our first concern consists in eliciting the user and the application that submitted the request. MSF uses a proxy server, thus the information under the client IP can not be used in user identification. Also, the log entries do not have information about which application submitted each request. The only information available is the log entries specifying the opening of an application: "`GET /{nameOfTheApplication}/index.action`".

- *Time-based heuristic.* We start with the time-based heuristic. Even though the aim of the experiments is not to find the best threshold value, we perform the experiments for two different thresholds (5 and 15 minutes), to see if their values make any significant changes in the sessioning of the log entries. If the timestamps of the requests after an application opening have a difference of less than the defined threshold, then they are considered part of the same session. Otherwise, they are discarded, as we cannot know the application making the requests. Using the example in Listing 4.1, the requests after the opening of App2 are considered part of the same session if $t_4 - t_3 \leqslant 5min(15min)$ and $t_5 - t_4 \leqslant 5min(15min)$, or discarded otherwise. On the other hand, these requests may comply with the threshold for App1 as well ($t_4 - t_2 \leqslant 5min(15min)$, $t_5 - t_2 \leqslant 5min(15min)$). As a result, requests that may come from App1 session, may be assigned to the next session, possibly increasing this way the error rate in two directions: not including the right log entries in Session1, and including the wrong ones in Session2.

Listing 4.1: Time based heuristic on MSF logs

```
1. GET App1/index.action        t1 Session1
2. request from App1            t2 Session1
```

```
3. GET App2/index.action        t3 Session2
4. request from App1 or App2    t4 Session1 or Session2
5. request from App1 or App2    t5 Session1 or Session2
```

As seen from the situation in the log snippet in Listing 4.1, regardless of the timestamps, we cannot be sure about the session of the requests after two applications open simultaneously. It is worth pointing out that the uncertainty would still persist, even if the information about the client's IP address was in the log file, as the same user could open several applications at the same time.

We can see in Table 4.2 (first two rows), the overall number of sessions (for sessions with more than 3 requests) in the data, the average sessions' duration, and the average sessions' size (number of requests in a session). We are not further analyzing these metrics, to see which of the thresholds is more performative, as from the interpretation, the time threshold, used in isolation from other fields, is not enough for identifying sessions. But we will compare them with the metrics derived from the results of the navigational-based heuristic.

Table 4.2: Statistics for the defined sessions

Heuristic	**No. of sessions**	**Avg. duration**	**Avg. size**
time 5 min	15,804	110 sec	63
time 15 min	15,937	127 sec	63
time 5 min, navigation	8,233	266 sec	122
time 15 min, navigation	6,586	413 sec	152

- *Navigational-based heuristic, combined with time constraint.* Next, we reconstruct the sessions using not only the time difference between the requests but also the referer. We use this heuristic for two different timeout thresholds. Extending the same example with referer information (see Listing 4.2), we can see that the session identification accuracy is straightforward when different applications are being consumed (from the same user or different ones). Log entries 4 and 5 are assigned to the right session, due to the information provided by the referer. (The value of refA is ignored, as the request body itself gives the application in use.)

Listing 4.2: Navigation based heuristic on MSF logs, different application

```
1. GET App1/index.action   t1 refA  Session1
2. request from App1       t2 App1  Session1
3. GET App2/index.action   t3 App1  Session2
```

```
4. request from App2      t4 App2  Session2
5. request from App1      t5 App1  Session1
```

We cannot say the same when the same application is being used by different users (see log snippet in Listing 4.3). Log entries 4 and 5 may belong to Session1 or Session2, but they may end up in the wrong session, due to the lack of client IP address information.

Listing 4.3: Navigation based heuristic on MSF logs, same application

```
1. GET App1/index.action   t1 refA Session1
2. request from App1       t2 App1 Session1
3. GET App1/index.action   t3 App1 Session2
4. request from App1       t4 App1 Session1 or Session2
5. request from App1       t5 App1 Session1 or Session2
```

As seen from Table 4.2 (two last rows), the metrics from the navigation method are significantly different from the ones when only the time heuristic was used (two first rows). We can see that after using the referer information, for the same logs, we have less number of sessions, but larger ones in terms of the number of requests and session total duration. This means that, when using only the time heuristic, we are over-splitting the sessions, thus potentially losing sequences of requests. Besides this, the new sessions created, likely contain mixed requests from different users and different applications, thus possibly creating fake sequences of requests.

Even though we applied grounded methods in pre-processing the logs, we admit that challenges like session identification and field extraction may still remain due to the lack of important information in the logs or the format being used. These problems are hard to deal with, and the best way to address them is to mitigate the root cause.

Assessment. The performance of the heuristics could be evaluated by comparing the constructed sessions with the real ones. In WAPI usage logs we cannot have the real sessions. Thus, in order to evaluate the accuracy of the reconstructed sessions of logs from MSF, we compare them and the reconstructed sessions of logs from WIDP, in the context of four specific applications. Different from MSF, the logs from the WIDP has information about the client IP addresses, but not the referer (Table 4.1). Consequently, we reconstruct the sessions using client IP and timeout (15 minutes). Then for each application, we extract the distinct requests assigned on both instances (Table 4.3). We saw that when using only the time heuristic in MSF, even though the sessions are on average shorter in terms of the number of requests (Table 4.2), too many distinct and different requests are assigned to each application. The same happens with WIDP, whose logs do not have information about the referer. For each application, we explore in detail the distinct requests assigned to them, for all the sessions. We saw that there were WAPI requests, that even though not related to the applications, were assigned to them because of the missing information in the logs.

Table 4.3: WAPI requests assigned to four applications installed in MSF and WIDP

Application	MSF (time)	MSF (time, navigation)	WIPD (time)
dhis-web-event-capture	106	28	177
dhis-web-event-reports	139	25	118
dhis-web-tracker-capture	202	48	124
HMIS-Dictionary	136	31	57

As the examples show and the experiments' results support, not being able to identify the users and the applications submitting the WAPI requests, greatly affects the correctness of sessions' identification. We can see an improvement when the referer information is available in the logs, but its exploitation comes with extra pre-processing efforts.

6 Common WAPI logs issues

In this section, deriving from the case study, we introduce the main WAPI usage logs quality issues, that are responsible for the problems that surged during field extraction and session identification. Some of them are related to the nature of WAPI usage logs, hence providers should be aware of and consider them before analyzing the logs. Others originate from the way WAPI usage is logged, and thus can be eliminated or ameliorated.

1. **Field extraction**

- Fields' separators part of the fields' body.

  Description: Even though it is recommended to use more human-readable formats for logs, keeping them machine processable is also important. Each log entry consists of several fields, typically separated by specific characters, e.g., comma, semicolon, and space. With the help of these separators, providers can perform the fields' extraction. Problems may arise when the fields themselves contain in their body the characters used as separators. The most heterogeneous fields are the request, the referer, and the user agent.

Impact: Not addressing this issue may result in extra added time and effort in log data pre-processing. If the special separators are part of the field's body, the automation of field extraction will generate errors, and providers should perform manual work to fix the issue.

Mitigation: To facilitate the field extraction, it is recommended to double-quote the fields that might have special characters like request, referrer, user agent, etc. The use of machine parseable formats will increase the automation of pre-processing, and therefore its correctness.

2. **Session identification**

- Insufficient fields.

  Description: WAPI usage logs are usually not logged for the purpose of analyzing consumers' behavior and getting indirectly their feedback. Thus, they often suffer from missing crucial fields for the application of several analyses, or other fields whose presence may enrich the analyses with new insights. We encountered this issue with the log data from WIDP, whose format did not include the referer.

  Impact: The lack of specific fields may become an impediment to applying several analyses or may affect the accuracy of the analysis results. The *referer* header is a field that contains the address of the page that made the request. Even though this is an optional field, it contains important and helpful information to reconstruct the sessions. As demonstrated in the introduced case study, in cases where: (i) the session identifiers are not present, (ii) the client IP address is actually a proxy address, and (iii) session timeout differs between several consumers' applications, the information under referer will help the analysts to better identify the session of a log entry and reconstruct the requests' sequence. Even though providers cannot fully rely on the referer information (it is not in the log when consumers type the request in the browser), it increases the correctness of assigning every log entry to its own session. The *user agent* field, which contains information about the browser and the operating system used when making the request, may as well play an important role in distinguishing the requests coming from different users, thus it should be included in the log format.

  Mitigation: In order to strike a balance between not leaving out important fields, and at the same time not logging too many fields, providers should decide beforehand on the analyses they will perform on the usage data, and the question they seek to answer. The specific requirements of the analyses should help them in making the right decision. Nevertheless, regardless of the type of analysis, providers should be able to identify different users and applications and log the needed fields accordingly.

- Missing applications' identifiers.

  Description: Applications' identifiers are unique identifiers that providers generate for their consumers, usually to monitor their usage for billing purposes. Consumers must include their applications' identifiers in each WAPI request so that providers can track their WAPI usage. Currently, this practice (of providing application identifiers) is typically followed by providers that have monetized their WAPIs. However, it can be used for more than just correctly charging consumers. We are not covering this in the 'Insufficient fields' issue, as, more than a field to be included in the log format, it is related to the providers' decision to generate this kind of identifier. Consumers may then submit the identifiers as an HTTP header, as a query parameter, or as a request body field.

  Impact: The lack of applications' identifiers is not likely to impact the accuracy and correctness of analysis results, but it affects the evaluation and the prioritization of the found usage patterns. Suppose that providers will find in the logs specific usage patterns that may indicate the need for some changes in the WAPI. Not knowing which applications are making the requests, providers cannot be sure whether the patterns found are coming from several applications, or from a few applications with a lot of users. In this situation, if they decide to perform the indicated changes, they will not know how many applications these changes will affect. To make informed decisions about the implementation of the prescribed changes, they should have information about these identifiers. Furthermore, under the conditions where sessions' identifiers are missing, application identifiers will help providers in improving the sessioning of the usage logs. We faced this issue with logs from both MSF and WIDP, as the DHIS2 provider was not generating these identifiers.

  Mitigation: Providers can address this issue by generating unique identifiers for each consumer's application so that consumers include them in all the requests made to the WAPI. Additionally, in order to differentiate between usage logs created during the development/testing phase and production phase, providers should generate different identifiers for each of the phases. As already explained, these usage logs manifest different aspects of consumers' behavior. Thus, providers should be able to separate them in order to accurately apply purpose-specific analysis.

- Hidden client IP address.

  Description: The client IP address gives the IP addresses of the applications' users. Combined with other information (session timeout, referer, user agent, etc.) this information can be used in users' identifications, as well as sessioning. However, if the consumers are using proxy servers, as in the case of MSF, the IP address that appears in the usage logs will not be of the original user doing the request, but that of the proxy server address. As a result, different users may appear under the same client (proxy) IP address in the logs, misleading the user identification process. We want to note that, as we are analyzing the way applications are consuming the WAPIs, our interest in users' identifications is limited to their help in sessions' identifications. By this means, if the same users appear with different IP addresses each time they use an application (dynamic IP addresses), this will not affect the analysis results.

Impact: Not being able to distinguish the requests from different users may produce mixed-up sequences of requests. The impact can be even more severe if other identifiers (e.g., application identifiers) are also missing in the logs, as in the example in Table 4.4.

Table 4.4: Log entries from different users with the same IP address (proxy address).

Client IP	Request	Timestamp	Referer
IP1	request1	18/Dec/2020 09:35:27.723	app1 domain
IP1	request2	18/Dec/2020 09:35:28.112	app1 domain
IP1	request3	18/Dec/2020 09:35:33.009	app2 domain
IP1	request4	18/Dec/2020 09:36:07.545	app3 domain
IP1	request5	18/Dec/2020 09:36:36.225	app1 domain

Mitigation: WAPI providers cannot control or fix this issue. Thus, it is important for them to be aware of this problem and not fully rely on this field for user and session identification. Instead, they should make sure to include other fields in the logs (e.g., referer, user agent), which will help them better structure the logs. This was the case with usage logs from MSF. While the client IP addresses were not usable, the referer information helped in logs sessioning.

- Timestamp coarse granularity.

  Description: The timestamp field shows the exact time the request was made to the WAPI. Even though the logging system stores the timestamp when the request was made, the log entry that represents that request is printed in the log file after the WAPI sends the response to the consumers. This means that the requests are not completely chronologically ordered in the WAPI usage file: a log entry printed after another one, may have been submitted earlier, thus it should have an earlier timestamp. As the difference, in this case, may be in milliseconds, if the timestamps are not logged precisely enough, the log entries may appear with the same timestamp

Table 4.5: Log entries with wrong order because of coarse timestamp.

Client IP	Request	Timestamp (as appears in the logs)	The real time
IP1	request1	18/Dec/2020	18/Dec/2020 16:05:55.824
IP1	request3	18/Dec/2020	18/Dec/2020 16:05:55.912
IP1	request2	18/Dec/2020	18/Dec/2020 16:05:55.859

(as in the example showed in Table 4.5), making their ordering unreliable [14]. This was the case with the logs from MSF. The timestamp granularity was in seconds, thus several requests had the same timestamp. We tried to fix this issue by keeping at least the order that the requests had in the file, and for every two consecutive requests with the same timestamp, we added one millisecond to the latter one.

Impact: Having the requests in the wrong order may adversely affect analyses' results, by producing erroneous usage patterns, and also hiding important ones.

Mitigation: To be able to order log entries exactly in a chronological way, providers should log the timestamp with high precision (e.g., milliseconds).

We have summarized in Table 4.6, the mitigation suggestions, based on the main problem they aim to solve. Actually, the potential errors derived from them can affect not only the log pre-processing but also the analyses, resulting in erroneous usage patterns. Thus, to assist providers in enhancing the usage logs of their WAPIs, we introduce this set of suggestions, that will not just help them to remedy the issues' effects on the data, but uncover and mitigate their root causes.

Table 4.6: Issues' mitigation for a better WAPI usage logging

WAPI usage log issue	Mitigation
Field extraction	Use a machine parse-able format for logs
Session Identification	Provide application identifiers
	Provide different application identifiers for development phase
	Log the referer, user agent
	Log the timestamp in high precision

7 Conclusion and future work

In this paper, we first show the potential of WAPI usage logs, by describing several analyses that providers may perform. Since the success of analyses strongly depends on the quality of their input data, we report the main issues of WAPI usage logs. We then conduct a case study to show the importance of the right log format. Next, derived from the case study, we identify a set of issues that may be present in these logs, explain how these issues impact the analyses, and suggest how to mitigate them.

Our results indicate that WAPI usage logs contain invaluable information about consumers' behavior, needs, and difficulties. But this beneficial information comes at the cost of the logs' tedious pre-processing. Typically, WAPI usage logs suffer from several issues, that should be properly addressed or mitigated, in order for them to be further analyzed. While some of these issues are related to the nature of the communication between WAPI and its consumers, others may occur because of improper logging. Furthermore, there are many demanding analyses, whose requirements should drive providers in the way they log the usage of their WAPIs.

In future work, we plan to perform the proposed analyses on the WAPI usage logs, applying first the suggestions in mitigating the existing quality issues.

Chapter 5

A Data-Driven Approach to Measure the Usability of Web APIs

The paper has been published in *Proceedings of Euromicro Conference on Software Engineering and Advanced Applications (2020)*
DOI: http://dx.doi.org/10.1109/SEAA51224.2020.00021

Abstract

Application Programming Interfaces (APIs) are means of communication between applications, hence they can be seen as user interfaces, just with different kinds of users, i.e., software or computers. However, the very first consumers of the APIs are humans, namely programmers. Based on the available documentation and the "ease of use" perception (sometimes led by corporate decisions and/or restrictions) they decide whether to use or not a specific API. In this paper, we propose a data-driven approach to measure web API usability, expressed through the predicted error rate. Following the reviewed state of the art in API usability, we identify a set of usability attributes, and for each of them, we propose indicators that web API providers should refer to when developing usable web APIs. Our focus in this paper is on those indicators that can be quantified using the API logs, which indeed reflect the actual behavior of programmers. Next, we *define metrics for the aforementioned indicators, and exemplify them in our use case, applying them to the logs from the web API of the District Health Information System (DHIS2) used by the World Health Organization (WHO). Using these metrics as features, we build a classifier model to predict the error rate of API endpoints. Besides finding usability issues, we also drill down into the usage logs and investigate the potential causes of these errors.*

1 Introduction

Application programming interfaces (APIs) represent the abstraction layer built upon sets of low-level methods and functions, in order to make them easily reused by third parties [70]. They are a means of communication between applications. Thus, we can say that APIs are user interfaces, just with different users in mind, meaning software or computers [12]. But we should not exclude from API users the human dimension.

Actually, the very first consumers of the APIs are humans, namely programmers. They are the ones who decide to use or not a specific API in their applications (sometimes under some corporate decisions and restrictions, e.g., pricing). If developers want to build a mobile application, and one of the features that they want to add to their application is the user location, usually it is up to them which location API to choose: Google Map API, OpenStreetMap API, Bing Maps API, Foursquare API, etc. Usually, API consumers decide to use an API by reading its documentation and by trying to perform different small tasks with it [12]. So, if API providers want to increase their customer outreach or the number of users of their API, they should focus on improving the API documentation and its easy-to-use interface.

In this paper, we propose a data-driven approach to measure API usability, based on how API consumers perceive and use APIs. Throughout our work we focus on web APIs, which differ from the traditional ones (statically linked APIs) mostly in the way providers and consumers are connected (via the internet, typically by HTTP protocol) or how consumers adapt when APIs change (if web API providers decide to disconnect an older version of web API, its consumers are forced to upgrade to the newer versions) [99], [27]. We use the API usability taxonomy proposed by Mosqueira-Rey et al. [64], which is based on the work of Alonso-Rios et al. [5]. We adopt this taxonomy to describe the usability of web APIs. As we explain in more detail in Section 3, there are some usability sub-attributes, which can not be investigated from the logs (they are related to the API source code and not to the interface). Therefore, for (almost) each of the usability sub-attributes, we propose some indicators that API providers should refer to. Based on these indicators, we define some metrics to measure each of the attributes. Then, we assess the relevance of these metrics in evaluating the usability of web API (reflected in the error rate of API endpoints), by building a classifier model that predicts the kind of error rate of endpoints based on the computed metrics.

We aim to not only find usability issues but also to investigate their root causes by drilling down into the usage logs. Consumers' behavior is imprinted in these logs, so their monitoring and analysis are crucial as they can play an important role in revealing usability issues. Since log data are semi-structured and often noisy, we explain how to perform the pre-processing step before doing the analysis.

We introduce and further use the following concepts:

- API resources - the data that API provides.
- API endpoints - the location where the resources can be found.
- API elements - resources, parameters, schema attributes.

Here we focus only on the interface level of the APIs. In other words, we evaluate API usability abstracting from the implementation code and the functionalities that API offers. Putting all together, our study is driven by the following research questions:

RQ1: Which are the usability sub-attributes that mostly influence the API consumers' experience?

RQ2: Can we find issues impacting these usability sub-attributes by analyzing the API usage logs?

RQ3: Can the usability issues found in the API usage logs be measured in a meaningful way from the API consumer point of view?

RQ3.1: How to carry out the pre-processing of API log data before performing usability analysis?

The main contributions of the paper are as follows:

- We perform an empirical study in measuring the web API usability, by monitoring and analyzing the API usage data.
- We define and adapt a set of measurable indicators for web API usability attributes, and quantify in terms of API usage log traceability the indicators of one of the usability attributes, know-ability.
- We perform an experimental validation of the importance of these metrics, by applying our approach in a real-world case study.

The remainder of this paper is organized as follows: in Section 2 we present the related work on API usability evaluation. In Section 3 we frame our approach. We describe the API log data pre-processing phase in Section 4 and present our case study in Section 5. We discuss our findings in section 6, and conclusions and future work in Section 7.

2 Related Work

The evaluation of API usability has gained a lot of attention in recent years [75], [66]. Based on the methods used, we can see two main categories: works that analyze the APIs design and their structural metrics, not taking into account how the API is being used (analytic methods [73], [80], [21]), and works that study how the API users are using the API (empirical methods [29], [33], [71]). We give a general overview of these works here, while the relevant usability attributes that we use in our study are explained in Section 3.

Rama et al. [73] presented a set of structural metrics that can be evaluated on the API interface, or on the API implementation source code. Scheller and Kühn [80] also provided several metrics to measure the usability of APIs. They conducted a literature review to identify factors that affect API usability and investigated in-depth a few of these factors. De Souza and Bentolila [21] provided a visual representation of APIs' complexity based on the complexity metrics of Bandi et al. [10].

Gerken et al. [29] on the other hand, used the concept map method to study and evaluate API usability. But, their longitudinal approach can only be applied to long-time segments. McLellan et al. [61] used the think-aloud protocol. They gave API code examples to four programmers and asked them to analyze and understand the code. They found usability testing very effective, based on the usability issues identified by the participants. Thus, they suggested iterative API redesign and testing phases. Piccioni et al. [71] designed an empirical usability study by interviewing 25 programmers and giving them a concrete task to accomplish using the API under study, in order to compare the participants' expectations with their actual performance. Grill et al. [33] evaluated API usability by applying a methodology of three phases: a heuristic evaluation (based on Zibran et al. [105]), a developer workshop and interviews, following this way both analytic and empirical methods.

Actually, as Rauf et al. [75] stated in their work on systematic mapping of API usability studies, the most used methods to evaluate API usability were empirical ones: usability tests, controlled experiments, surveys, etc. In fact, these methods are the most consolidated in software usability or human-computer interaction (HCI) studies. As APIs differ from regular traditional software (can be used in scenarios that even API designers have not thought about before, are prone to frequent changes, etc.), their usability evaluation requires more automated, scalable, and time-efficient methods [75].

Few studies focus on mining repositories. Zibran et al. [105] studied five different bug repositories to identify the most reported API usability issues (by the API consumers). More than one-third (37.14%) of the bug reports in their study were related to API usability. From these, the most frequent ones were missing features, correctness, and documentation. Macvean et al. [56] analyzed specifically the usability of web APIs. They took data from Google API Explorer to identify APIs, with which developers struggle more and spend more time and effort in learning and using. The metric they used to measure API usability was API request error rate (client-side erroneous requests (4XX) per total requests to the API). Nonetheless, they recognized that other metrics can also be considered to better evaluate the usability, like API consumers' satisfaction measured by API surveys or the number of erroneous requests consumers make until they achieve a successful call (known also as the time to first *hello world* or *ah ha moment* [30]). Although their results were still preliminary and, as they stated, early in nature, the methodology used seems promising and opens a lot of areas for future research. Murphy-Hill et al. [65] developed a tool that analyzed consecutive snapshots saved by developers to derive problems of the API. They checked the API methods that developers changed between snapshots, but more than the 'worst' usability problems, these changes reflected the most used API. They suggested the use of other heuristics, like analyzing consumers' experience, comparing the experience of novice API consumers with the more familiar ones, etc.

Mosqueira-Rey et al. [64] used both analytic and empirical methods. They adopted a general usability framework from Alonso-Rios et al. [5]. They derived from it a set of heuristics and guidelines for traditional APIs and used these to evaluate the usability of a given API. Nevertheless, they pointed out the need to expand their work toward web API. Indeed, we use and follow their taxonomy that comprises six top-level attributes, refined into 21 more specific sub-attributes, to give and define a set of indicators that web API providers should use to offer usable web APIs.

As Wittern et al. [99] stated in their work, there is still a lack of research on the consumption of web API (strongly related to usability). As already annotated, web APIs differ from traditional ones, hence, new methods need to be used in evaluating their usability, as well as other assumptions need to be taken into account. To understand and analyze the web API consumers' behavior, we do not conduct observational or controlled experiments, but instead monitor and analyze the web API log data, which indeed contains the interaction between API and its consumers.

Table 5.1: API usability

Usability Attributes	Sub-attributes	Indicators	Traceability in the logs
1.Know-ability	1.a Clarity	API elements' name clearness, descriptiveness, unambiguity, similarity [105], [65], [80], [73], [71]	✓
	1.b Consistency	Uniformity in naming API elements [105]	✓
	1.c Memorability	The number of API endpoints' parameters [105], [73], [80], [10]	✓
	1.d Helpfulness	The use of different status codes for different situations [105], [73]	✓
2. Operability	2.a Completeness	Consumers workaround solutions for the missing features [16]	✓
	2.b Precision	Proper data types to avoid loss of precision and unnecessary type-casting [5, 33, 73, 80, 105]	✗
	2.c Universality	The use of universally recognized names, formats, etc., for API elements [64]	✓
	2.d Flexibility	Multiple ways to do the same thing [71, 105]	✓
3. Efficiency	3.a In human effort	The balance between the flexibility of having different ways of doing a task and the complexity of having too many options [71]	✓
	3.b In task execution	API response time [30]	✓
	3.c In tied up resources	The excessive use of shared resources made by API [64, 105]	✗
	3.d To economic costs	The excessive costs required for the API use [64]	✓
4. Robustness	4.a To internal error	The proper handling of internal errors [80, 105]	✓
	4.b To improper use	The proper handling of consumers errors [80, 105]	✓
	4.c To third party abuse	The handling and mitigation of abusive behaviour of third party	✓
	4.d To environment problems	The handling of errors coming from environment problems	✓
5. Safety	5.a User safety	The use of safe HTTP methods to change resources	✓
	5.b Third-party safety	The confidentiality protection of the users' personal information [64]	✓
	5.c Environment safety	The security of the users' assets [64]	✗
6. Subjective satisfaction	6.a Interest	The trend of API users over time (API new consumers, API churn rate) [30]	✓
	6.b Aesthetic	The aesthetic of API elements' name (no weird names or special characters used in an inappropriate way) [64]	✓

3 The proposed approach

In this section, we describe the key elements of our approach. We start by explaining the different types of API logs and which usability attributes can be evaluated using each of them. Then we introduce the usability attributes, sub-attributes, and indicators inferred and adapted from the literature review. Last, we interpret these indicators in terms of API usage log traceability, where possible.

3.1 Measuring web API usability in web API logs

Web API usage logs can be collected at the provider side, at the consumer side, or at proxy servers [45, 84]. The usage data collected in each case reflect different aspects of the API usage. Data collected from the consumer side have all the requests made to the API, but only from that consumer. Log data from different consumers should be gathered to have more generalizable results from the analysis. On the other hand, the data logged on the API provider side contain information about all the consumers of the API, but if consumers have adopted API response caching, they do not record the requests for which the responses are cached. Moreover, development logs cannot be distinguished from production logs [56].

Nevertheless, the information transmitted through development logs differs from that in production logs. The former logs are created while the developers are creating and testing their applications. We can see the developers' learning curve as well as their difficulties while using the APIs, only on development logs [56]. On the other hand, production logs are created during the post-development phase of the applications, after the applications are launched and used by their end users. The analysis of these logs can give us insights about API new usage scenarios, not evident even to API providers. Here we can address issues related to all the other usability aspects.

All in all, we cannot evaluate all the usability attributes in one type of API log: different API logs are needed to evaluate different attributes. For example, we cannot measure the completeness of an API, if we have development logs from one API consumer. We need production logs from different API consumers to conclude if the API in the study lacks some features, forcing this way its consumers to come up with different workarounds.

3.2 API usability aspects

API usability represents a qualitative characteristic [75]. As such, there exist different interpretations, different terminologies, and different definitions [5, 75]. After choosing the usability taxonomy to expand for web APIs, we performed a literature review in order to quantify each of the sub-attributes into indicators.

Initially, we searched for works on API usability assessment, for both traditional and web APIs. As in our study we measure the usability based on the API interface, we filtered out the works that focused their analysis on the API implementation code. Next, we consolidated into indicators for each sub-attribute, all the information gathered. Then, considering that most of the information was for traditional APIs, we adapted it for web APIs. For example, Rama et al. [73] mentioned in their work as structural metric the one related to exception classes "Using exception throwing classes that are too general with respect to the error conditions that result in exceptions". We link this with the status codes returned and give as indicator the use of different status codes in different situations so that web API consumers would know the exact error that happened. We classify this as an indicator of the helpfulness of the API. Finally, there were some sub-attributes for which we could not find any pertinent information in the literature review, thus we extend the current state of the art with additional indicators. Table 5.1 summarizes our findings and reports main usability attributes with their sub-attributes, and their indicators, and points out if such indicators can be traced in API logs.

3.3 Usability issues detected in API usage logs

As already stated, we evaluate API usability by analyzing API usage logs. Due to different server setting parameters, logs may comply with different formats, but typically each log entry has information about the client's IP address, the request time, the request method (GET, POST, etc.), the request body, the protocol, the time needed to respond to the request, the status code, and the size of the object returned.

We evaluate from the API logs those attributes and sub-attributes that can be mapped to the information in the log entries (see Table 5.1). Thus, we focus on indicators consisting of the information about the interface of the API (naming of API elements, number of parameters) and indicators about the interaction consumer - API (status codes, duration, request sequences).

As a matter of fact, not all the usability sub-attributes of the taxonomy can be evaluated based on the information in the logs. For example, the precision of the API, an operability sub-attribute, is mostly related to the precision of the data types used. Data type selection is seen as too critical. API consumers should not perform type casting when it is not necessary, as this will not only increase their effort but also affect the precision [5,33,73,80,105]. However, in the log entry, requests are just strings, so we cannot analyze the parameters' data types (part of the implementation code of the functions and methods under the APIs).

Know-ability implies the ease of understanding, learning, and remembering the API. Among others, this attribute is mainly related to the naming of the API elements. To properly evaluate the naming, it is essential to take into account the purpose and the functions of the API elements, and then perform semantic analysis of their names. In automated solutions, this is almost infeasible [80]. Hence, the controls that we perform for clarity, consistency, and memorability sub-attributes are channeled in names' similarity, the style of naming, path/query length, query complexity features, etc. On the other hand, helpfulness is related mostly to accurate documentation and detailed error messages. In the logs, this can be manifested in the erroneous repeated requests.

Operability is mostly associated with the API consumers' needs fulfillment. Tracking workarounds built by consumers when API is not offering them a direct solution is quite difficult. Anyway, consumers' interaction with the API is imprinted in the API logs, so from there we can infer behaviors that address this issue. Part of operability is also universality, which implies the use of universal names and symbols. Next, flexibility is a double-edged sword: for experienced programmers, it is considered beneficial, but for novice ones, it increases the complexity of APIs (as explained below).

Efficiency can be evaluated in terms of human effort, time, and resources spent while using the API. Regarding human effort, usually, the more complex the API is, the more effort is spent from the consumers' side. Complexity is a very general concept and comprises several usability attributes. As having several ways to do the same thing sometimes confuses the programmers [71], we consider this as an indicator in evaluating efficiency according to human effort. Efficiency in task execution is reflected in the time that an API needs to respond to a request, which in the logs is stored as duration. For the costs of using the API, we look at the API endpoints, that can be optimized regarding the number of calls. Consumers have to pay for the number of calls for non-free APIs. So, if there are endpoints that are always called one after the other, their merging can reduce the consumers' expenses. We do not have a log-based indicator for the efficiency in tied-up resources sub-attributes, as this is not related to the API interface, and is not reflected in any log entry fields.

Robustness, defined as the property of an API to handle errors and adverse situations, is strongly related to the status codes that APIs send to their consumers. Even though applications that consume the API should also be robust and treat properly the error situations, APIs should not fail in front of incorrect or even improper use. Therefore, APIs have to handle both the errors that come from their side (5xx errors) and the errors from the consumers' side (4xx errors).

Safety deals with the challenge of mitigating risks or damage while the consumers are using the API. Consumers typically access web APIs using HTTP requests. APIs should not allow the consumers to change resources using safe HTTP methods (methods that do not modify resources, e.g. GET). The safety of the APIs is also associated with third-party safety, as well as API environment safety. For the last one, we do not have a log-based indicator.

Subjective satisfaction is the capacity of the API to engage its users and preserve their interest. API providers can evaluate this by monitoring the trend of new API consumers. Additionally, the aesthetic of API is related to the aesthetic of API elements' names.

4 API log data pre-processing

An API log file contains the requests made to the API. This information is raw, so before applying any analysis technique, the data should undergo a pre-processing phase (see Figure 5.1), typically counted as the most difficult task in the Usage Mining process [84, 89]. While it usually consists of three steps, namely data fusion, data cleaning, and data structuring, we include a fourth step, data generalization [89]. See the final steps in Figure 5.1.

Data fusion. We already explained part of the data fusion step discussing different types of API log data (see Section 3.1). Based on where we collect the data (consumers' or providers' side), we extract the log files and merge them (if they are on different servers). Before proceeding with the pre-processing steps, the data might be anonymized, because log files might contain information considered sensitive (identifiable personal information). One way of handling this concern is by masking the sensitive data (i.e., IDs, or IP addresses) with surrogate identifiers and then proceeding with the next steps [89].

Data cleaning. This step mainly consists of removing irrelevant and noisy data from the files and correcting the data by means of adding or completing missing values. When fusing data from several sources, the logs from different sources can have different formats. Thus, when merging them, we may need to adapt their formats: adding/removing certain fields, dealing with quoted/unquoted fields, etc. On the other hand, whether to keep or remove certain log entries depends much on the purpose of further analysis [89]. For example, if one wants to discover user session profiles in web log data, he/she should filter out: (i) the log entries that result in errors, (ii) the log entries that have a request method different from GET, and (iii) the ones that access image files [39]. If the purpose of the analysis is to support caching or pre-fetching, then log entries for accessing images should not be excluded from the analysis [89]. Since we aim to measure the usability of the APIs, the status code is one of the most valuable fields in the log entries. The error rate of each API endpoint can reveal usability issues that can highly affect API consumers. Therefore, for purpose of measuring the usability of the APIs, we keep the erroneous requests and filter out from the file, the log entries that are not API requests, and the ones that do not imply API resource manipulation. These are some typical examples, but we certainly do not exclude the possibility of having to remove other log entries, depending on how the information is logged. After this step, the number of log entries will be highly reduced [84, 87, 89].

Data structuring. This step includes user and session identification. By users

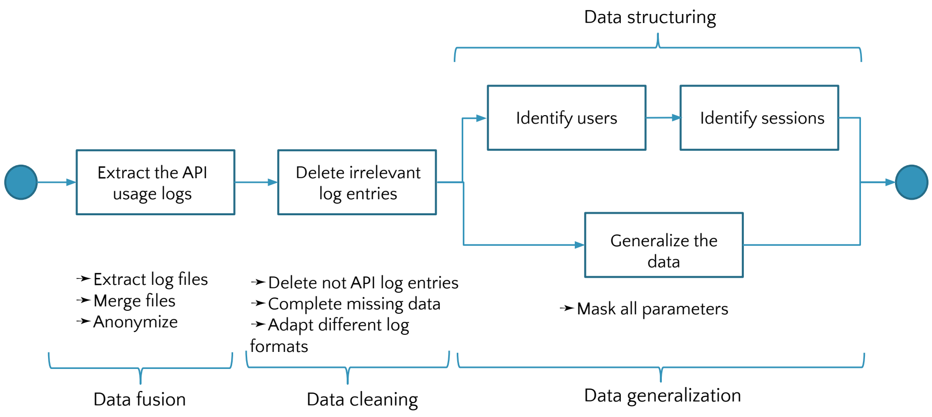


Fig. 5.1: Data pre-processing.

here we mean the applications that are consuming the API, so this step should be performed when working with API logs from the provider side. Ideally, when an application uses an API, the logs generated should have an ID that identifies its pathway with the API. Usually, the log file contains only the device's address (i.e., IP) and the user agent (i.e., software agent like a browser or an email reader). When applications' identifiers are not in the API logs, each IP might be counted as a user [89]. On the other hand, log data are usually not completed with the session ID, hence the sessions' identification results especially challenging, due to several reasons (i.e., caching, proxy servers, the same device used by several users, etc.) [89]. Thus, the sessions must be inferred by combining available user identification and approximate timeout simulating the time spent by a single user using the API.

Data generalization. This step is considered an advanced pre-processing step, comprising one of the most complex tasks in API log data analysis [87, 99]. It consists of extracting general API specifications from the requests in the log files. For instance, if we have "`https://.../api/country/Spain/regions`", the challenge would be to detect "`/country/`" and "`/regions/`" as resources' name (fixed part of the path) and "`/Spain/`" as a parameter value (dynamic part of the path). Synthesizing general API description from API usage is a hard problem, and existing solutions are hampered by the API logs nature (noisy, incomplete) and also API design and implementation problems [87, 99].

5 Case study design

5.1 DHIS2 Web API

We analyzed the log data of the District Health Information Software 2 (DHIS2) web API. DHIS2 is an open-source, web-based health management information system platform used worldwide for data entry, data quality checks, and reporting. It has an open REST API, used by more than 60 native applications. External software can make use of the open API, by connecting directly to it or through an interoperability layer.

DHIS2 is as well instantiated as WHO Integrated Data Platform[1] (WIDP) at World Health Organization (WHO), and is used by several departments for routine disease surveillance and country reporting. For the analysis, we use API log data from the development instance of WIDP, with more than 50 applications installed, core or built in-house. The logs date from September 2018 to November 2019.

5.2 Data pre-processing

We instantiate the data pre-processing workflow previously introduced in Section 4and further discuss the challenges encountered in different steps of the process.

Data fusion. As previously mentioned, we had the log data from WIDP, which is a DHIS2 API consumer. But as several applications are installed on this platform, it partly behaves as the provider. The logs were recorded using a customized Apache log format, which contained the following information: client IP, request long date (date, time, timezone), duration (time needed to send the response), keepalive, request (method + resources URL + protocol), response code, the size of the object returned and the user agent.

Data cleaning. We discarded the log entries with request method HEAD or OPTIONS, as they do not imply any resource manipulation or resource retrieval [87]. We filtered out also the log entries that were not related to APIs, thus not containing the “/api/” entry point, predefined to be in the API request. These are usually log entries for the loading style files, fonts, or graphics. But at the same time, we were careful not to remove key log entries (i.e., log entries about the login, logout, and applications’ first access) that, even though did not contain the “/api/” entry point, played an important role in data structuring. After data cleaning, our log file contained 2,268,291 log entries (i.e., requests), out of the 5,936,203 that it had initially.

[1]`http://mss4ntd.essi.upc.edu/wiki/index.phptitle=WHO_Integrated_Data_Platform_(WIDP)`

Data structuring. For user identification, we used the information under "client IP" in the log entry to identify different API consumers (i.e., users). We considered as a session all the requests made by one user (client IP), with a time difference of no more than 15 minutes. We assigned incrementally a number to each session as an identifier, ending up with 40,067 sessions, from 849 users. As we were analyzing the know-ability of the API, our focus was not on the applications that were using the API (i.e., the real consumers of API), but on the programmers (i.e., the first consumers of the API). However, if measuring, for example, the completeness (investigating workarounds) of an API, one should identify which application submitted each request, in order to be able to build an exact sequence of the requests for each application. As already mentioned, in our case, the log files contained log entries from each of the applications installed in the platform. But the log entries did not have information about which application submitted each request. The only information we had was the log entry specifying the opening of any application: "`GET /dhis2-dev/{nameOfTheApplication}/index.action`". But the same user could open several applications at the same time. So, from the moment the same user (ClientIP1) opened more than one application, we could not be sure about the origin of the next log entries:

```
ClientIP1 "GET /{App1}/index.action"
ClientIP1 "requests from App1"
ClientIP1 "GET /{App2}/index.action"
ClientIP1 "requests from App1 or App2"
```

In these analysis scenarios, we might need to make approximations and combine IP information, the current app(s) opened, and the timeout. The lack of application identifiers can impede not only user identification but also hinder data cleaning.

Apart from user and session identification, we performed also "target endpoint" identification. In order to be able to aggregate statistical information for each endpoint (total number of requests, requests with client-side error, etc.) we assigned a "target endpoint" to each request that got a client-side error response code. For example, if we want to compute statistical metrics for the endpoint "`/api/organizationUnitGroupSets`", then we should somehow match it with the endpoints: "`/api/organization-unit-group-sets`" or "`/api/organizationsUnitsGroupsSets`", which have a wrong syntax, but the programmer intent was the first endpoint. We started by extracting all the requests that got a status code not related to syntax errors. Using the Levenshtein distance algorithm [51], we computed the similarity of these endpoints with each real endpoint in the requests body and grouped together the most similar ones.

Data generalization. During this step we had to define which parts of paths were fixed (i.e., resources) and which were dynamic (i.e., parameter values). That is, for several endpoints with path body like "`api/userGroups/uNJOBaIw/users/H4atNsEr`", or "`api/userGroups/BzbYRSpk/users/D78WJM8J`", we inferred from them a general one, with generic API specifications "`api/userGroups/ID/users/ID`". We applied some ad hoc procedures, and masked all the parameter values with the same string "ID".

5.3 Data Analysis

We assumed that an API that suffers from poor know-ability, will have a high error rate. For each sub-attribute, based on the defined indicators, we computed the below metrics and created a model to predict the kind of error rate.

- **Clarity:** We analyzed the endpoints' names and the similarity between them, assuming that similar names confuse users and increase the chances of making errors. We expect that endpoints with a higher similarity will have more client-side error response codes. We split each "target endpoint" into its elements. For each of them, we found the most similar one, by computing their similarity. For example, the `account` resource had the highest similarity of 0.71 with `count`, `constants` 0.82 with `constraints`, and so on. Using this information, we then computed two metrics for each endpoint: the **average similarity** and the **maximum similarity**. For example, `userDataStore/gridColumns/eventCaptureGridColumns` has three elements, `userDataStore` with similarity 0.69, `gridColumns` with similarity 0.48, and `eventCaptureGridColumns` with similarity 0.72. So the endpoint average similarity coefficient will be 0.63, while the maximum similarity will be 0.72, coming from `eventCaptureGridColumns`.

- **Consistency:** We focus on the syntactical aspects of naming to evaluate the consistency. Thus, we analyzed the **naming style** of endpoints (names that contain only lower case or numbers, upper cases, underscores, hyphens, special characters, or more than one of these "styles"). Actually, we do not expect a specific naming style to be the cause of errors from the client side. We will investigate the impact that the existence of several naming styles, can have on API consumers. Clearly, the semantic aspects (use of synonyms, homonyms, etc.) are also very important when we analyze the consistency of an API and can be evaluated using different tools for natural language processing. We will include this in our future work.

- **Memorability:** We analyzed the path part as well as the query part of the requests. First, we computed the **path length** as the overall number of characters; and stored under **path elements** the information about the number of elements in the path (e.g., "`analytics/events/aggregate/ID`" has three elements in the path body). Then, we examined the query part to quantify its complexity. We measure the **query length** as the number of characters; we analyze the query syntax and impute the **transformation** metric as the number or transformation functions in the query; we represent the **logical operators** as the number of logical operators used; we defined the **query depth** as the number of nested objects, giving to each nested object the same weight, despite the number of fields it contains, as we are analyzing the complexity of the query part from the programmer point of view (amount of code to be written) and not the machine point of view (amount of time to compute the results of the query); we counted the **query parameters**; and also the **schema attributes** used in each request. Both are part of the query length, commonly used to evaluate query complexity, but as they represent different ways of reducing the result size, we choose to count them separately. As we realized that consumers often use the same query parameter or schema attribute several times in a single request, we decided to reflect this also in different metrics: **unique query parameters** and **unique schema attributes**.
- **Helpfulness:** To measure this sub-attribute, we analyzed the endpoints that got repeated client-side error codes from the same user. We assume that the lack of details in the error messages and the lack of examples in the documentation can lead consumers to repeat the same mistakes. Therefore, we grouped all the requests with the same "target endpoint", from the same client IP, and analyzed those that had 2 or more client-side errors. We imputed the **error repetition rate** as the number of errors per total number of requests for the same endpoint for each programmer.

Besides the metrics per each request, we computed the error rate as the number of erroneous requests per all requests. We first selected only those endpoints that were in requests with an error rate greater than zero and divided them into two classes: endpoints with an error rate higher than 0.3 were considered with poor usability and the ones with an error rate lower than 0.3 with no usability issues. There were 1128 endpoints with certain values of the metrics. In order to balance the class distribution to 40% for poor usability and 60% for no usability issues, we randomly extracted endpoints with an error rate of zero to obtain the desired proportion.

First, in order to reduce over-fitting and facilitate the interpretability of results, we performed attribute (i.e., metrics) selection, keeping only those metrics that were more relevant for predicting the class. We run the CorrelationAttributeEval with Rank method on WEKA[2], which ranks the attributes by measuring the correlation between them and the class. From all the aforementioned metrics, maximum and average similarity, query depth, logical operators, and path length were the more relevant ones.

Then, to see if we could predict the class based on these metrics, we built a decision tree using WEKA implementation of J48, with the default parameterization and 10-fold cross-validation. We used a classification model because we were interested in finding if the API had or not usability issues, more than predicting the exact error rate (as a regression model would imply). Thus, using the selected attributes from the information in the logs, we obtained a model able to predict the class of the endpoints with an accuracy of 72.25%. Considering that in our analysis we do not take into account other factors in API usability like API functionality, the semantics of API names, API programmers' experience, etc., we aimed at high precision, more than at a high recall. But, even though we were not aiming to find all the endpoints with usability problems only from the data in the API logs, our model performed quite well with a recall of 0.722 (Table 5.2, 5.3).

Table 5.2:
Confusion Matrix

a	b	classified as
272	280	a
103	725	b

a=poor usability
b=no usability issues

Table 5.3:
J48 Results

Correctly Classified	72.25%
Incorrectly Classified	27.75%
Kappa statistic	0.389
Mean absolute error	0.399
Weighted Avg Recall	0.722
Weighted Avg Precision	0.723

6 Discussion

To get finer insights into the gained classification, we analyzed in more detail the branches of the decision tree. We found that requests that did not have a query part tend to have a lower error rate, even when the path had an average similarity higher than 0.61. The error rate was also low for those endpoints with both average and maximum similarity low, respectively lower than 0.61 and 0.75. On the other hand,

[2]https://www.cs.waikato.ac.nz/ml/weka

the error rate was high for those endpoints that even though had a low average similarity, they had at least one element in their path with very high similarity. Endpoints with high average similarity and a query part had also a high error rate.

Additionally, we examined closer the endpoints with a higher error rate, to see which were the most common mistakes done by the programmers. We point out the following issues:

- Consumers try different name styles until they find the right one. The fact that in the same API, different resources are named using different styles, confuses them. For example, before typing `userRoles/ID`, one of the consumers tried with `userrole/ID` and `user-roles/ID`.

- We encountered another consistency issue in the naming of API, plural and singular form of resources' names. As some of the resources' names were in the plural form, and others in the singular form, consumers tried their different forms. For objects with composed names (i.e., multi-word names), the error rate increased. For example, before typing `organisationUnitGroups`, one consumer tried three different versions: `organisationUnitGroup`, `organisationUnitsGroups` and `organisationUnitsGroup`. The lack of consistency in naming resources decreases the memorability of the API. Different naming conventions (pascal or camel case, underscore, etc.) and the semantic nature of the analysis needed were the main reasons why these two aspects (plural/singular form and multi-word names), were not reflected in any of the metrics.

- When analyzing the repeated errors for the same endpoints from the same consumers, to measure the helpfulness, we noticed low memorability perception from consumers. From 1,283 cases of repeated client-side errors, 659 of them had a non-client-side error response code for the first request. This implies that even after understanding the API and submitting correct requests, consumers fall into mistakes.

Threats to validity. The main threat to construct validity involves the arbitrarily chosen threshold of 0.3 for the error rate class. To mitigate this, we plan to redefine the threshold by conducting other use cases (i.e., independent datasets). This way, we will also minimize the external validity threat, whose main concern is related to the one API we have as the use case.

7 Conclusion and Future Work

We reviewed the current state of the art in API usability evaluation in order to identify those usability attributes that mostly affect the programmers' experience in using the APIs. We combined the gathered information and for each usability attribute, we specified an indicator that web API providers should use to provide usable web APIs. We embodied the indicators for the know-ability attribute into several metrics, which we later computed using the web API usage data from our case study. In order to assess the significance of these metrics, we built a classifier to predict the kind of error rate based on the endpoints' specifications. Know-ability issues that more influenced the API consumers' experience were more related to similar API elements' names, multi-word names, and lack of consistency in naming convention.

In future work, we plan to define more metrics for the attributes under study and expand our analysis to other usability attributes. In order to redefine the threshold and the generalizability of our model, we will take under study other use cases. On the other hand, to evaluate the analysis results we plan to conduct a supplementary empirical analysis, directly asking the API users for their opinion on whether they encountered usability issues while using the API or not.

Chapter 6

Web API Evolution Patterns: A Usage-Driven Approach

The paper has been published in *Journal of Systems and Software (2023)*
DOI: `http://dx.doi.org/10.1016/j.jss.2023.111609`

Abstract

As the use of Application Programming Interfaces (APIs) is increasingly growing, their evolution becomes more challenging in terms of the service provided according to consumers' needs. In this paper, we address the role of consumers' needs in WAPIs evolution and introduce a process mining pattern-based method to support providers in WAPIs evolution by analyzing and understanding consumers' behavior, imprinted in WAPI usage logs. We take the position that WAPIs' evolution should be mainly usage-based, i.e., the way consumers use them should be one of the main drivers of their changes. We start by characterizing the structural relationships between endpoints, and next, we summarize these relationships into a set of behavioral patterns (i.e., usage patterns whose occurrences indicate specific consumers' behavior like repetitive or consecutive calls), that can potentially imply the need for changes (e.g., creating new parameters for endpoints, merging endpoints). We analyze the logs and extract several metrics for the endpoints and their relationships, to then detect the patterns. We apply our method in two real-world WAPIs from different domains, education, and health, respectively the

WAPI of Barcelona School of Informatics at the Polytechnic University of Catalonia (Facultat d'Informàtica de Barcelona, FIB, UPC), and District Health Information Software 2 (DHIS2) WAPI. The feedback from consumers and providers of these WAPIs proved the effectiveness of the detected patterns and confirmed the promising potential of our approach.

1 Introduction

Application programming interfaces (APIs) provide an abstraction layer built upon sets of low-level methods and functions. API providers build these methods and functions and offer their interfaces so that API consumers (i.e., software developers) can easily use them in their applications. While applications locally access the 'traditional' APIs (i.e., statically linked APIs), web APIs (WAPIs) are exposed over the Internet, hence their endpoints (URLs to WAPI resources) are remotely accessed.

Being provided, and thus accessed remotely, imposes challenges for both parts, notably during WAPI evolution. Consumers are obliged to update their applications under WAPI evolution pace [97], [53], [26], [27]. On the other hand, providers have their own burden: striking a balance between not imposing unexpected, frequent changes and providing an up-to-date, maintainable, bug-free WAPI, that fulfill consumers' needs and business (organizations that are exposing the data) requirements. Knowing the considerable impact WAPIs have on consumers, providers would benefit from their feedback to better understand their needs [65]. Consequently they can implement changes directly targeting consumers' concerns.

If consumers encounter any bugs, or they need a new feature implemented in the WAPI they are using, they commonly report the bug or request the new feature to WAPI providers by means of issue tracker systems, communities of practice (e.g., `https://community.dhis2.org`) or other communication channels between providers and consumers (explicit feedback given to providers). However, due to the lack of these communication channels or because of consumers' neglect to report, several WAPI bugs remain unreported, and several desired features or improvements, unlisted and undiscovered [47].

Actually, as WAPIs are exposed over the network, providers can observe the way consumers use them, and can indirectly track consumers' needs. Furthermore, they can detect flaws related to the usability of the WAPIs, of which consumers may not be easily aware. For instance, in large companies with separated WAPI development teams for each domain (i.e., for marketing, sales, product, and customer service), it may happen that, for the same functionalities, several WAPI endpoints are designed.

The way consumers access them, may reveal their redundant, duplicated, or combinable functionalities.

We consider consumers' implicit feedback as the fuel of this process. Assuming that WAPI evolution is and should be driven by the way WAPI is consumed, we propose a data-driven approach to anticipate changes based on consumers' behavior. Therefore, consumers' behavior is recorded in these logs, and their analysis can find room for potential improvements, or obliquely reveal consumers' needs for new features hidden under several workarounds (solutions found by WAPI consumers that allow them to get data, functionality, or features they need, but that are not yet implemented by providers). There are several WAPI monitoring tools available, but they are mostly oriented toward providing reporting dashboards or automatic alerting in case of WAPI failure [23]. Consequently, WAPI providers have all this potentially insightful, large volume of data that is being generated, but not proactively used for evolution.

Here is where process mining promises to be useful, providing a set of well-established analysis techniques to extract process-related insights from event logs [92]. By making use of the WAPI usage data available on the providers' side, and the applicability of process mining in web services context [91], [93], we can analyze the way consumers use the WAPI, and identify behavioral patterns (usage patterns that indicate specific behaviors) that can imply the need for change. Additionally, having all the usage scenarios from consumers for the endpoints that will be modified, providers can provide better migration scripts to help them migrate to new WAPI versions.

This paper is in the same line as our previous works, where we target WAPI evolution and leverage the availability of WAPIs usage logs. Previously, we analyzed WAPI development usage logs (logs generated when consumers are developing and testing their applications) [42] and prescribed a set of usability-related changes. Then, we focused our research on production logs analysis (generated after consumers' applications' release) and presented our preliminary results through a tool that aims to support providers in planning the changes [44].

Building upon these previous results, in this paper we extend the goal of [44] and elaborately introduce a method to help providers in the evolution of their WAPIs. We keep the position that WAPIs evolution should be mainly usage-based, i.e., the way consumers use WAPIs should be one of the main drivers of WAPI changes [3]. To this aim, we first characterize the relationships (e.g., consecutive calls) between two endpoints (URLs to access API resources). Specific relationships may hint several behavioral aspects, indicating room for potential improvements (e.g., merging endpoints), the presence of workarounds, or other reasons for performing changes to the WAPI. We summarize these relationships into a set of behavioral patterns whose occurrences suggest the need for specific WAPI changes. Then, we analyze the usage logs to elicit the actual use of the WAPI and detect the occurrences of the pre-defined patterns. Employing process mining techniques, we detect from the logs snippets of the entire process model with WAPI-related semantics, in the form of several metrics for the endpoints and their relationships. We use these metrics to later detect frequent patterns.

The main contributions of the paper are as follows:

- We define and present a set of WAPI behavioral patterns, whose occurrences indicate the need for changes.
- We introduce several useful metrics following process mining techniques in order to detect the occurrence of patterns in the logs.
- We demonstrate the applicability of our approach by exemplifying it on real-world usage logs from two different WAPIs: (i) District Health Information Software 2, and (ii) Barcelona School of Informatics at the Polytechnic University of Catalonia (Facultat d'Informàtica de Barcelona, FIB, UPC).
- We show the significance of the detected patterns and the feasibility of the proposed changes, by interviewing the WAPI consumers and providers to directly assess the results.

2 Background

In this section, we introduce the primary concepts used throughout this paper. We start with describing the main input of our approach, namely the WAPI usage logs, over which we apply process mining techniques. We explain some of their general definitions and then interpret them in the WAPI case. After that, we introduce graphlets, a concept that helps us in specifying the structural relationships between endpoints, which we later use to define the set of patterns.

2.1 WAPI usage logs

Providers typically log their WAPIs' usage by recording all requests done against the WAPIs. Every time a consumer issues a request to a WAPI, a log entry is generated and stored in the usage log file. The information that is logged for each request and its format may vary due to different logging system setting parameters and also providers' decisions about logs design. For example, using the Apache

Custom Log Format[1] (a flexible and customizable format), when an application makes a request like `https://api.fib.upc.edu/v2/sales?client_id=ID` to an WAPI endpoint, the following information could be logged by providers (based on the logging system configuration): the IP address of the application's user, the timestamp when the consumer made the request, the request method (GET), endpoint ($v2/sales?client\_id = ID$), the protocol (HTTP/1.1), the time needed to respond to the request, the status code (e.g., 200 if the request was successful), the size of the object returned, the address of the page that initiated the request, information about the operating system or browser used, and other fields comprising different aspects of consumers' applications and their users (e.g., `10.28.120.115 [24/Sep/2018:16:00:03 +0200] GET v2/ sales?client_id=ID) HTTP/1.1 116 200 9436 "https://..." "Mozilla/5.0...")`.

Based on consumers' applications lifecycle, we distinguish two different types of WAPI usage logs: (i) development logs generated at design time, and (ii) production logs generated at runtime. We point out this distinction as each of these logs owns specific characteristics and elements of interest, and can be used to analyze consumers' behavior from different aspects.

Development logs are generated while developers build and test their applications. Thus, these logs show their attempts in using the WAPI, the endpoints they struggle more with, or specific mistakes they make while using and learning the WAPI [42], [56]. As developers may freely try different requests, several times, and pose any query, the analysis of these logs gives insights about the usability of the WAPI.

Conversely, production logs are generated while applications are being used by end-users. Since the applications are released for final use, it is assumed that they are quite stable, thus the WAPI requests predetermined by the implemented functionalities of the applications are less error-prone. By analyzing these logs, providers may identify consumers' needs for new features, and implement the corresponding endpoints. These logs can reveal new usage scenarios providers may have not thought about, instructing them in including these scenarios in the documentation. Besides these, providers may identify ways of improving the WAPI based on how consumers use it, merging endpoints that are always called together for a specific purpose, removing endpoints that are not being used or that are superseded by other endpoints, or creating new endpoints derived from the existing ones that are always called with specific values for some parameters (e.g., the new COVID-19 endpoint that Twit-

[1]`http://httpd.apache.org/docs/current/mod/mod_log_config.html`

ter released based on the high interest researchers had for the conversations on this topic[2]).

As the main goal of this work is to identify the need for changes based on the way consumers use the WAPIs in their applications, we make use of the production logs.

2.2 Process Mining

Process mining is a process-oriented data mining discipline that uses event logs to extract process-related insights like building the process model, detecting discrepancies between the model and the monitored behavior, or improving the model based on the monitored behavior. We give the definition of the fundamental concepts of process mining, as presented in [92], and adapt them to the WAPI domain:

- *Activity* - a specific step in the process. For WAPIs, the set of activities is the set of WAPI endpoints and consumers' actions (i.e., request methods like GET, POST, PUT to the endpoint).
- *Event* - an activity occurrence at a specific time. We refer to WAPI requests as events and identify them by activity names and timestamps.
- *Case* - a process instance. We refer as a case to a set of requests that an application is submitting to the WAPI during a certain time period. In the web domain, a case is commonly referred to as a session. Each event belongs to a specific case.
- *Event log* - a set of cases. In the WAPI domain, the event log corresponds to the WAPI usage log, which contains all the requests (i.e., events) that consumers (i.e., applications) make against the WAPI.

In order to apply process mining, the WAPI requests in the usage log should have at least three attributes: (i) a case identifier that uniquely identifies the case to which each event belongs (composed of one or more columns), (ii) activity name that characterizes each event, and (iii) timestamp that indicates the time when an event occurs. The presence of these three attributes allows us to infer process insights from the event log. Certainly, other attributes that might store additional information about the events (e.g., application ID, status code), whenever available, add value to the analysis.

Typically, the output of process discovery is a process model, which can be a process map, a graph-based model, or other representations that describe the actual process based on the generated logs. In the WAPI context, process discovery consists on mining the usage logs to create the process model, where nodes represent

[2]https://twittercommunity.com/t/new-covid-19-stream-endpoint-available-in-twitter-developer-labs/135540

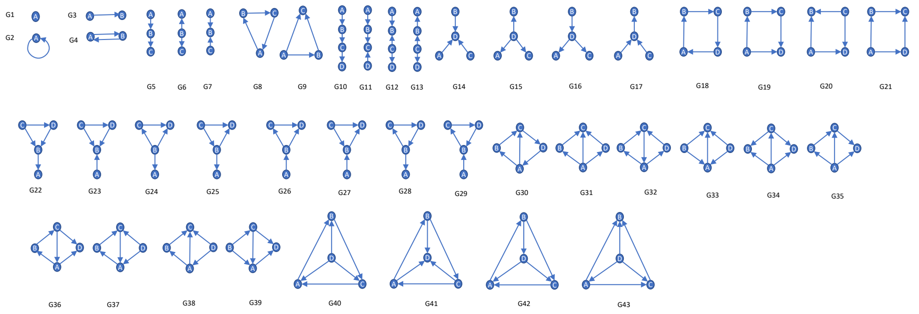


Fig. 6.1: Directed graphlets with up to 4 nodes [79]

the endpoints being called and directed edges the calling sequence of two endpoints. However, WAPIs are a typical example of less structured processes (they do not define a strict sequence of execution, but rather offer interactive functionalities like sharing data or passing messages [34]): consumers access the resources according to their needs, without following a strictly defined sequence of calls. As a result, the discovered process models turn out too complex, and consumers' behavioral patterns can hardly be identified. Even though there exist several techniques that tackle the simplification of complex models (i.e., clusterization, prioritization of activities and paths using several metrics such as fitness and precision, or significance and correlation as in fuzzy miner algorithm [20], [34]), these kinds of simplifications may affect the accuracy of the identified behavior. As an example, we can mention the sequences with direct consecutive calls that can be found in the simplified process models. These sequences heavily rely on the level of simplification applied during the model creation. Hence, sometimes, the direct consecutive call sequences shown do not really exist as the less frequently called activity in the middle of two more frequently called activities may be removed. To avoid identifying inaccurate patterns, we are not opting for any simplification of the complex spaghetti-like process. In fact, the complexity of the end-to-end process model does not limit, nor strongly affect our approach, because we are interested in the narrow relationships between a limited number of endpoints (2 to 4 endpoints), likely called together to perform a specific task. Therefore, we focus on the zoomed-in views of the end-to-end process model and put particular attention and interest in how specific nodes are related to each other.

2.3 Graphlets

To find all the possible ways the endpoints can be connected with each other and to characterize the structural relationships between them, we make use of the concept of graphlet. Graphlets represent small, connected, and non-isomorphic sub-graphs used to characterize and compare the structure of larger networks by detecting their local structure properties [72]. There exist several graphlet-based methods that exploit the information encoded in large networks by counting all the graphlets of these networks. Generally, graphlet-based methods for analyzing directed networks, as in our case, do not consider graphlets with more than four nodes, due to the composed nature of their structure (i.e., subsumption of smaller graphlets) and also for computational reasons [79].

Fig. 6.1 depicts the set of all possible directed graphlets composed of up to four nodes. Nodes represent WAPI endpoints, while edges represent their calling order: an edge from node A to node B shows that endpoint B was called after endpoint A. The general notion of directed graphlet, as defined in [79], includes graphlets from two to four nodes, and does not contain the ones that connect two nodes with edges in opposite directions. In order to represent the possible consumers' behaviors, we added the graphlets $G1$ and $G2$ representing a graphlet with only one node, and $G4$ as the graphlet with edges in opposite directions.

3 Approach

In this section, we give a general overview of our approach and the main steps we follow to identify the need for potential changes from the WAPI usage logs. It allows analyzing the usage logs in an iterative fashion. Thus, after implementing the detected changes, providers can monitor how consumers are adapting to them. Then, based on consumers' new behavior (reflected on the newly generated usage logs), providers may introduce new changes.

As our main objective consists in suggesting WAPI changes based on the extracted behavioral patterns, we need to specify beforehand which specific behaviors may reveal different consumers' needs or may be an indication for change. Indeed, instead of blindly searching for any usage patterns, we prune the search space to the patterns expressing consumers' needs for change. Therefore, different from techniques that mine the process model and extract from it frequent patterns [38], [48], [18], in our approach, we pre-define the set of patterns for which we look into the logs.

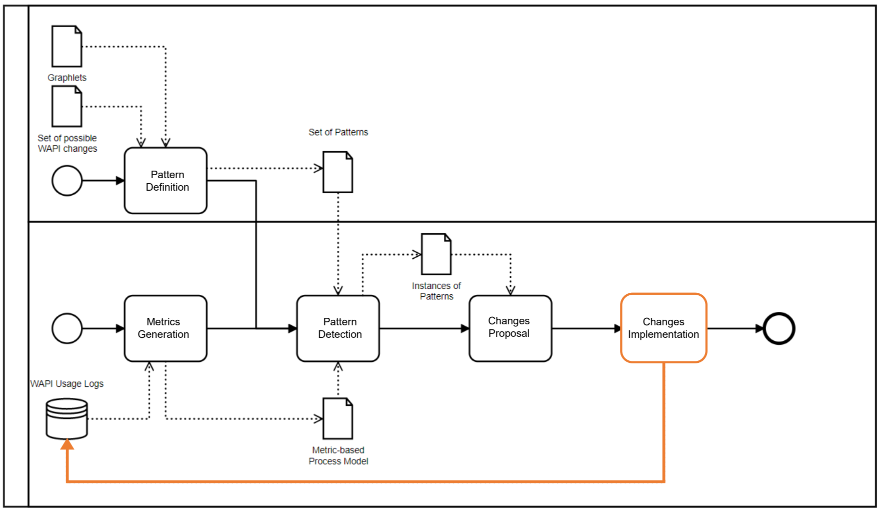


Fig. 6.2: Main steps of the approach

Fig. 6.2 depicts the main steps of the approach, which consists of two crucial stages. The first stage (illustrated in the upper part of the figure), which comprises the *'Patterns Definition'* step, can be performed every time providers want to detect different users' behavior through specific patterns. If providers are interested in following other types of behavior (not only those that indicate the need for change), they can include them in this first step, and then proceed with the other steps accordingly. In our case, we search the graphlets that encode structural relationships between endpoints representing changes that can happen to WAPIs according to [41], to define this way the set of patterns. We explain this step in more detail in Section 4.

The second stage (illustrated at the bottom part of the figure) can be performed multiple times, periodically when more usage logs are generated. During *'Metrics Generation'* step, we analyze the WAPI usage logs using process mining, build the order in which the consumers call the WAPI endpoints, and, for each endpoint, we calculate several metrics, to obtain a metric-based process model. Having the set of patterns and the metric-based process model, we detect the occurrences of patterns in the usage log (*'Patterns Detection'*). We associate a change to each pattern and suggest them to the providers (*'Changes Proposal'*). Although some WAPI usages may be interesting but no frequent, we focus on those usages that are frequent, as the changes they imply would bring more improvements. As one endpoint might appear in several patterns, domain knowledge should be used to prioritize the patterns and evaluate the feasibility of the changes' implementation. WAPI providers can properly evaluate the patterns, and decide if the indicated changes can be implemented (taking into account not only consumers' usage but also business requirements or maintainability issues, which cannot be observed from the logs). If providers decide to implement the changes (*'Changes Implementation'* is an external step done by providers, thus shown in orange in Fig. 6.2), consumers have to use the new WAPI endpoints. The logs generated after the consumers adapt to the new changes will show if consumers are using the WAPI endpoints as expected. Even though analyzing the new-generated logs is out of the scope of this paper (shown with the orange arrow in Fig. 6.2), it poses opportunities to not only review consumers' adaptation to the changes but also to measure the improvement brought by the changes. This indicates that our approach seamlessly works whether it is its first iteration or it is being applied after implementing some of the previously proposed changes. If introduced in the development lifecycle of WAPI, our approach can eventually converge if the usage is stable/constant, but it can also react and detect occasional changes in the behavior (new data requirements, new types of users/applications, etc.) and propose an adaptation of the WAPI to this new behavior.

In summary, the study presented in this paper answers the following research questions:

RQ1: In which ways can endpoints be related to each other to indicate the need for change based on how consumers call them? We extract from the given graphlets all the possible ways endpoints can be connected and then summarize in a set of behavioral patterns those relationships whose occurrences in the logs may indicate room for improvements or suggest the need for specific WAPI changes.

RQ2: What information can we extract from usage logs, that can help in detecting the defined relationships between endpoints? We introduce the set of metrics needed to identify the occurrence of the defined patterns. These metrics' values will be extracted and calculated from the usage logs.

RQ3: How can we detect the occurrence of the patterns in the usage logs? We expound the detection of the defined set of patterns, using the introduced metrics.

RQ4: To what extent can we propose changes based on the patterns found in the usage logs? After detecting the occurrence of the patterns in the logs, we evaluate their significance, first from consumers' point of view (i.e., to see if the patterns manifest consumers' need for improvements), and second from providers' point of view (i.e., to see if the changes we propose are feasible and beneficial for providers). Hence, we divide this question into two subquestions:

RQ4.1: To what extent can we understand consumers' needs based on the behavior manifested in usage logs?

RQ4.2: To what extent can we interpret the identified consumers' needs into feasible future changes?

4 Defining and detecting the WAPI behavioral patterns

In this section, we describe in more detail the main steps of our approach. In Section 4.1 we answer RQ1 by defining the set of patterns. In Section 4.2 we answer RQ2 by introducing and describing the metrics we generate after applying process mining. Then, in Section 4.3 we answer RQ3 by exhaustively explaining the detection of the patterns. We answer RQ4 later in Section 5.

4.1 Patterns Definition

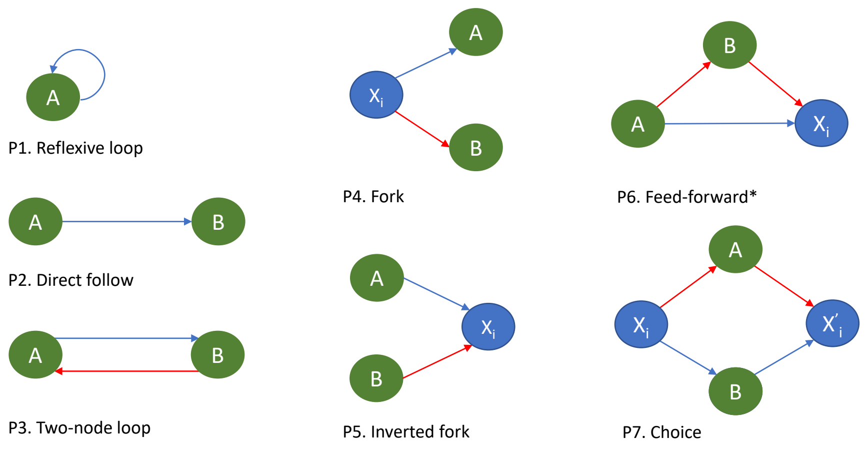
Using process mining concepts, we give the following definitions:

Definition 1. *A sequence is an ordered occurrence of activities within a case.*

Definition 2. *A pattern represents a relationship between activities and is inferred from frequent sequences that these activities are part of. Each pattern reveals a specific consumer behavior and might be an indication for change.*

In order to define all the structural relationships the graphlets represent in terms of sequences in the logs, we extract all possible sequences that can be generated from each of the graphlets of Fig. 6.1. In our case, there are no well-defined first or last-called activities. Thus, nodes that have at least one outgoing edge can be the first called endpoints of the sequences, while nodes that have at least one incoming edge can be the last endpoints of the sequences. To get the longest sequences from each graphlet, we start from nodes with only outgoing edges and finish with nodes with only incoming edges (if available).

Then, for every two endpoints, we analyze the sequences they are part of. If consumers' behavior manifested in these sequences (e.g., repetitive calls, consecutive calls) indicates the need for a change [41] (e.g., creating new parameters, merging endpoints), we classify the relationship as a pattern along with the change it indicates.



Edges with different colors belong to different cases. Green nodes represent the endpoints whose relationship the pattern describes. Blue nodes represent the endpoints that help detect the pattern.
[∗] The name for this pattern is adopted from Milo et al. [63], where the authors introduce several network motifs (specific sub-graphs) that appear frequently in different complex networks.

Fig. 6.3: The set of patterns

Even though the graphlets are built with unlabelled nodes (the nodes differentiate from each other by their position in the graphlet, meaning their incoming and outgoing edges), we are labeling the nodes to simplify the explanation of the generated sequences. It is important to note that even though we are interested in the relationships between at most two endpoints, other endpoints called by consumers close to these two play an essential role in identifying and detecting these relationships.

To illustrate how we elicit the patterns from the graphlets of Fig. 6.1, we give two extended examples:

- Let us look at graphlet G9 and list the sequences that it can generate. Node A has only outgoing edges, so this node will be the first called endpoint of the sequences. In the same way, node C has only incoming edges, therefore it will be the last called endpoint of the sequences. Thus, the sequences that this graphlet can generate are {(A, B, C), (A, C)}. Analyzing the relationship between endpoint A and B, the derived relationship from this graphlet is: an endpoint that is optional or complementary to another (i.e., B is optional between A and C). The same relationship can be found in G22, G23 (i.e., D is optional between C and B), etc. (more details in Appendix A).
- Let us look at graphlet G21 and list the sequences that it can generate. Node A has only outgoing edges, while node C has only incoming edges, meaning that

the generated sequences will start from A, and end in C. Thus, the sequences that this graphlet can generate are {(A, B, C), (A, D, C)}. The derived relationship is: endpoints that are possible choices between other endpoints (i.e., B and D called between A and C). The same relationship can be found in G31, G32, etc. (more details in Appendix A).

Same relationships can be derived from different graphlets. We summarize the identified relationships into seven WAPI behavioral patterns, as in Fig. 6.3. It is important to note that one endpoint may be part of more than one pattern occurrence, but the prioritization of the changes that the patterns indicate will be made using experts' domain knowledge.

The implied changes target WAPI improvement in different aspects: (1) reduce the number of WAPI calls that consumers have to make to get the needed data (and consequently reduce consumers' costs, in case consumers are paying-per-WAPI request) (P1, P2, P6), (2) decrease the traffic between applications and WAPI server (P1, P2, P6), (3) reduce the time needed to get the desired result from the WAPI (considering each request as an opportunity for consumers to make a mistake, and for WAPI provider to return an error message [62]) (P1, P2, P3, P6), (4) simplify the WAPI by getting rid of endpoints whose functionalities can be combined (P2, P3, P4, P5, P6, P7), (5) reorganize endpoints that serve the same purpose and supersede each other (P3, P4, P5, P7).

4.2 Metrics Generation

In this section, we respond to RQ2. We introduce a set of metrics, that can be quantified by extracting information from the logs.

To detect the occurrence of the patterns in the logs, we use two basic concepts from the association rules research field, support and confidence [4], and adapt them to the context of our approach. The *support* of an association rule of the form $X \Rightarrow Y$ is the fraction of instances where both the antecedent X and consequent Y hold. In our context, we refer as support to the frequency with which the instances of a pattern appear in the log file. For instance, given a pattern that describes a causal relationship between two endpoints (e.g., A being called implies B is called directly afterward), the support of this pattern will be equal to the frequency of the edge (A, B). For patterns formed with more than one sequence (e.g., A being called implies B can be called or C can be called), we take the geometric mean of the frequencies of the sequences that form it, (A, B) and (A, C) respectively. We do so to properly summarize the frequencies, and reduce the effect of skewed data (i.e., one of the two sequences has a considerably higher frequency than the other). We use absolute frequency (i.e., the total number of times that a sequence was called, not normalized by the size of the log file) as endpoints may be completely independent of each other, and the frequency of some endpoints with very high usage should not undermine the frequency of the other endpoints. Consequently, support values may not be in the same range.

The *confidence* of an association rule shows the fraction of instances containing the antecedent X, for which the rule $X \Rightarrow Y$ is satisfied. In our context, we refer as confidence to the number of times the pattern really occurs in the log, over all the possible times it could have occurred. We express it as the relative frequency of the sequences that form the pattern, regarding the frequency of the nodes whose relationship the pattern describes. For instance, to continue with the same example as in support, the confidence of the pattern describing the relationship between endpoints A and B (A being called implies B is called directly afterward), will be the frequency of the edge (A, B) (the number of times B is called directly after A), divided by the frequency of A (the number of times B could have been called after A). We subtract from the endpoint frequency the frequency of the endpoint self-loop (i.e., $freq^*_A = freq_A - freq_{AA}$), so this metric can better reflect the relationship endpoint A has with other endpoints. We use this pruned (from self-loops) frequency for all the patterns that describe the relationship of one endpoint with another endpoint, different from itself.

Typically, in process mining, for each node and edge, several frequency metrics are extracted from the logs (e.g., absolute frequency, case frequency, maximum repetition). As support refers to the frequency of the patterns, we adapt it for different frequencies, as follows:

- Case Support - the number of different cases in which a sequence (or an endpoint, i.e., a sequence of length one) appears. The case support of a pattern shows how the overall absolute frequency of the pattern is distributed in several cases. For instance, a pattern with a high absolute frequency but a low case frequency indicates that few cases are following the pattern many times.
- Average Case Repetition Support - the average number of repetitions of following a sequence within a case (the ratio of absolute frequency and case support).
- Maximal Case Repetition Support - the maximum number of repetitions of following a sequence within the same case.
- Application Support – the number of different applications that have called a sequence. It also reflects the number of applications that will be affected by the changes that the pattern indicates.

In general, one is interested on those patterns (rules) with interestingness metrics (e.g., confidence, support) above some minimum thresholds. As the values for these thresholds may depend on complexity and cleanness of the data, or even business requirements and objectives, they usually are set by domain knowledge holders. We choose not to arbitrarily decide on the values of these thresholds (which would allow us to present the precision and recall of the mined patterns), as it might affect the soundness of the results. Instead, we make a more qualitative evaluation of the patterns.

4.3 Patterns Detection

In the following of this section, we respond to RQ3. We describe each of the patterns, exemplify them to facilitate their understanding, and explain how we detect them in the logs.

Reflexive loops (identified in graphlet G2 in Fig. 6.1.) *Motivation:* The structure appears in the log as an endpoint called consecutively several times by a consumer, presumably with different values for the same attribute. It may point out a weakness in the endpoint design, that forces consumers to make unnecessary repetitive calls. Thus, it may indicate the need for a new parameter to which multiple values can be passed.
Example: Let us suppose that `GET api/events?date="15-12-2021"` returns all the events happening in `15 December 2021`. If consumers submit consecutive calls for different dates, then a useful optimization that reduces the number of WAPI calls is to change the parameter `date` (or to create a new one), so that an array or a range of values can be passed.
Detection: We calculate the metrics for the pattern as follows:

$$support = freq_{AA}, \qquad confidence = \frac{support}{freq_A}$$

In addition to these two metrics, for the reflexive loop pattern, we calculate the average case repetition support, the maximal case repetition support, and also the longest sequence of reflexive loops. For instance, let us suppose that for the reflexive loop of the endpoint `GET api/events?date="date_value"` we have the following figures: the reflexive loop appears on average three times per case (the sequence (A, A) is found on average three times in each case), maximum fifteen times in one case (the sequence (A, A) is found on average fifteen times in each case), and the longest sequence of reflexive loops is eight (endpoint A is being called nine times consecutively in at least one case). If the suggested change will be implemented, it will improve the WAPI by reducing the number of calls on average three calls per case (each of the three sequences of (A, A) will be replaced by one call of A) and on a maximum of fifteen calls per case (each of the fifteen sequences of (A, A) will be replaced by one call of A), by replacing two calls of the endpoint with just one. As the longest sequence of reflexive loops is eight (the endpoint called nine times sequentially), there will be cases where nine calls will be replaced with one.

Direct-follow nodes (identified in graphlets G3, G5, G10, etc., in Fig. 6.1.) *Motivation:* The structure appears in the log as two endpoints called consecutively after one another. This pattern may imply a causal dependency relationship between these two endpoints, which forces consumers to call them together in that specific order to fulfill a specific task. As such, we suggest the possibility of merging the two endpoints into one that combines the functionality of both of them. Based on the dependency between the two endpoints, we define two types of the direct-follow relationship: *(i) endpoint B called always after endpoint A,* meaning that endpoint B is the dependent one, and *(ii) endpoint A called always before endpoint B,* meaning that endpoint A is the dependent one. Both relationships are similar as they represent an immediate succession between endpoints. They differ only on the position of the dependent endpoint (the first called endpoint of the sequence, or the second one), and this is reflected in the way we detect the pattern and calculate its metrics.
Example: Let us suppose that `GET api/sales` endpoint returns the list of all the classrooms of the faculty building, and `GET api/sales/reserves` returns the list of the all the reservations of the classrooms. If we detect that several consumers frequently call these two endpoints one after the other (type (ii) of the direct follow the *sales* endpoint is followed by the *reserves* endpoint), then we assume that the information about the reservations complements the classrooms' information. Therefore, we propose the creation of a new endpoint, which will embody the functionalities of both endpoints.
Detection: We calculate the metrics for the type (i) of the pattern as follows:

$$support = freq_{AB}, \qquad confidence = \frac{support}{freq^*_B}$$

For the type (ii) of the direct-follow pattern, the metrics will be calculated as follows:

$$support = freq_{AB}, \qquad confidence = \frac{support}{freq^*_A}$$

As mentioned, the difference consists in the node, whose dependent relationship the pattern describes. In the first one, the dependent node is node B, thus the confidence measures how significant is the incoming edge (A, B) for node B. In the same way, the confidence for the type (ii) measures the significance of the outgoing edge (A, B) for node A, as A is the dependent node, after which only node B is being called. To obtain more insights regarding the occurrence of the pattern in the data, we calculate the other mentioned metrics (naming the average case repetition support and the maximal case repetition support).

Two-node loop (identified in graphlet G4 in Fig. 6.1.) *Motivation:* This structure appears in the logs as two direct follow patterns in opposite order, i.e. (A, B) and (B, A). It describes the relationship between two endpoints that are called together, but not necessarily in the same order. Thus, it might indicate that these two endpoints have combinable functions and the data returned by them might be reorganized to better fulfill consumers' needs. *Example:* Let us suppose that `GET api/program/local?name ="MIRI"` and `GET api/program/international?name="BDMA"` endpoints are being called one after the other, in different order. This behavior can be an indicator that separating the endpoints to get the data about the offered master programs (local master MIRI and international master BDMA), does not match consumers' needs, who want to know the information of several masters regardless of their condition. WAPI providers can decide to offer just one standardized endpoint for the master programs `GET api/masterprogram?name={MIRI, BDMA}` (i.e., merge the two endpoints), and create a new parameter by which consumers can filter the data according to their needs. It is worth noting that we are not questioning the design decision of having two separate endpoints instead of a unified one. But, if consumers of the WAPI access those endpoints in that specific way, we recommend the merge, as the changed WAPI will better fit its own consumers' needs.
Detection: We calculate the metrics for the pattern as follows:

$$support = \sqrt{freq_{AB} \cdot freq_{BA}}, \qquad confidence = \frac{support}{\sqrt{freq_A{}^* \cdot freq_B{}^*}}$$

Fork (identified in graphlets G6, G13, G15, etc., in Fig. 6.1.) *Motivation:* This structure describes the relationship between two endpoints that are called after the same set of endpoints. It appears in the log as sequences of calls that have as target one of the two related endpoints, and as source one from the common set of endpoints. This pattern implies that the two related endpoints might have highly similar functionalities or purposes. As in Fig. 6.3.4, after calling endpoints X_i, consumers call endpoint A or B. Providers may consider this usage as an indicator to reorganize endpoints A and B, and merge them into one single endpoint.
Example: Let us suppose that after calling endpoint X_1 `GET api/analytics`, consumers call endpoint A `GET api/charts` to get the data for visualizing them

in a chart form, or endpoint `B GET api/reportTables`, to get the data for visualizing them in a tabular format (we are using Fig. 6.3 nodes' label to simplify the example). These two endpoints, `A GET api/charts` and `B GET api/reportTables` are also being called after X_2 `GET api/dimensions` and X_3 `GET api/indicators`. This way of using the charts and reportTables endpoints (i.e., calling them after the same set of endpoints), may indicate to providers that these two endpoints can indeed be combined into a unified endpoint `GET api/visualization` with a new parameter by which consumers can filter the data according to their needs.

Detection: We calculate the metrics for the pattern as follows:
Let $X = \{X_1, ..., X_N\}$ be the set of all endpoints such as $(X_i, A), (X_i, B) \in L$, for $i = 1, \ldots, N$, where L is the log file:

$$support = \sum_{i=1}^{N} \sqrt{freq_{X_iA} \cdot freq_{X_iB}}, \qquad confidence = \frac{support}{\sqrt{freq_A{}^* \cdot freq_B{}^*}}$$

As in the two-node loop pattern, we take the geometric mean of the frequencies of the sequences from each X_i to A or B and sum them to calculate the occurrence of the pattern.

Inverted fork (identified in graphlets G7, G11, G14, etc., in Fig. 6.1.) *Motivation:* This structure describes the relationship between two endpoints that are called always before the same set of endpoints. It appears in the log as sequences of calls that have as sources one of the two related endpoints, and as target one from the common set of endpoints. As in the *Fork* pattern, this pattern implies that the two related endpoints might have highly similar functionalities or purposes. As in Fig. 6.3.5, after calling endpoint A or B, consumers call any of endpoints X_i. Providers may consider this usage as an indicator to reorganize endpoints A and B, and merge them into one single endpoint.

Example: Let us suppose that after calling the endpoint `A GET api/rooms`, WAPI consumers might call the endpoint X_1 `GET api/reservations` to get the reservation about the room, the endpoint X_2 `GET api/location` to get the location of the room, or the endpoint X_3 `GET api/map` to get the room's position on the map of the campus (we are using Fig. 6.3 nodes' label to simplify the example). The same three endpoints, X_1 `GET api/reservations`, X_2 `GET api/location`, and X_3 `GET api/map`, are being called after the endpoint `B GET api/laboratories`. As the rooms and laboratories endpoints have the same possible choices, providers might consider reorganizing the data that these endpoints return, differentiating the type of the space by a parameter.

Detection: We calculate the metrics as above: Let $X = \{X_1, ..., X_N\}$ be the set of all endpoints such as $(A, X_i), (B, X_i) \in L$, for $i = 1, \ldots, N$, where L is the log file:

$$support = \sum_{i=1}^{N} \sqrt{freq_{AX_i} \cdot freq_{BX_i}}, \qquad confidence = \frac{support}{\sqrt{freq_A{}^* \cdot freq_B{}^*}}$$

Feed-forward (identified in graphlets G9, G22, G23, etc., in Fig. 6.1.) *Motivation:* This structure appears in the log as two different sequences, where one is formed by three endpoints called in sequence (A, B, X), and the other sequence skips the second endpoint, hopping from the source node directly to the target node (A, X). This relationship may imply that the second endpoint's functionality (i.e., B) might be optional or supplementary for the first endpoint. Thus, providers might decide to join the functionality of the second endpoint to the first one, by creating a new parameter for it. To simplify the illustration of the pattern, we consider one endpoint X_1, but endpoints A and B may be both followed by more endpoints (i.e., X_i).
Example: Let us suppose that endpoint `A GET api/competences` returns the list of all the competencies required for any degree, endpoint `B GET api/competences_type` returns data about the type of the competences (e.g., technical competence, transversal competence), and endpoint X_1 `GET api/subjects` returns the data about the needed subjects for a degree (we are using Fig. 6.3 nodes' label to simplify the example). Let us suppose that after calling the endpoint for the required competencies for a degree some consumers call the endpoint for the needed subjects for the degree. On the other hand, some other consumers call the endpoint about the competencies type after getting the data about the competencies, and then the endpoint of the subjects. By including the information about the competencies type to the competencies endpoint, providers will not only enhance the functionality of the competencies endpoint, but at the same time, they will reduce the number of calls for the consumers who were calling it always with the competencies type endpoint. Even though providers' action might be the same as in other patterns (merging endpoints), the structure of the pattern and the way it appears in the log is different, thus its detection is different.
Detection: We calculate the metrics as follows: Let $X = \{X_1, ..., X_N\}$ be the set of all endpoints such as $(A, B, X_i), (A, X_i) \in L$, for $i = 1, \ldots, N$, where L is the log file:

$$support = \sum_{i=1}^{N} \sqrt{freq_{ABX_i} \cdot freq_{AX_i}}, \qquad confidence = \frac{support}{freq_A{}^*}$$

Choices (identified in graphlets G21, G31, G32, etc., in Fig. 6.1.) *Motivation:* This structure describes the relationship between two endpoints that are called always before and after the same set of endpoints. This structure appears in the log as sequences of calls of three endpoints, that differ by the second called endpoint: (X_i, A, X_i') and (X_i, B, X_i'). After calling endpoint X, consumers may call endpoint A or B, and then endpoint Y. This behavior implies that endpoints A and B, as called together with the same endpoints, might be substitutes, meaning sharing similar functionality, or alternatives to each other, meaning having different functions, but potentially cater to a similar purpose. Thus, this usage might be an indicator for providers to merge these two endpoints into a new one, that consumers can call with different parameter values depending on their needs.
Example: Let us suppose that after calling the WAPI endpoint X_1 `GET api/student/ID1` to get data about a specific student, consumers call the endpoint `A GET api/student/ID1/fails` to get the subjects where the student has failed, and then the endpoint X_1' `GET api/student/ID1/credits` to get the data about the credits the student has (we are using Fig. 6.3 nodes' label to simplify the example). On the other hand, other consumers after calling X_1 `GET api/student/ID1`, may call the endpoint `B GET api/student/ID1/passes` to get the data about the subjects the student has already passed, and then the endpoint X_1' `GET api/student/ID1/credits`. In this case, the providers might decide to merge the endpoints about the failed and the passed subjects into the same endpoint and filter the status of the subjects for a specific student using a new parameter.
Detection: We calculate the metrics for the pattern as follows:
Let (X, X') be the set of all pairs of endpoints such as $(X_i, A, X_i'), (X_i, B, X_i') \in L$, for $i = 1, \ldots, M$, where M is the set size, and L is the log file:

$$support = \sum_{i=1}^{N} \sqrt{freq_{X_i A X_i'} \cdot freq_{X_i B X_i'}}, \qquad confidence = \frac{support}{\sqrt{freq_A{}^* \cdot freq_B{}^*}}$$

5 Evaluation

In this section, we present the evaluation of our approach. In Section 5.1, we first introduce the two use cases in which we apply the approach. We tackle log preparation and its analysis in Section 5.2. In Section 5.3 we present the evaluation results.

5.1 Use case

In order to demonstrate and evaluate the defined patterns in real-world examples, we analyze usage logs from two WAPIs, belonging to two different domains.

The first set of usage logs comes from the District Health Information Software 2 (DHIS2) WAPI. DHIS2 is an open-source, web-based platform to implement health management, developed and maintained by the software development group within the Health Information Systems Programme at the University of Oslo (UiO), Department of Informatics. It is used worldwide by various institutions and NGOs for data entry, data quality checks, and reporting. It has an open REST WAPI, used by more than 60 native applications (consumers). For the analysis, we use logs from two different DHIS2 instances: (1) the World Health Organization (WHO), in their Integrated Data Platform[3] (WIDP), which is used by several WHO departments for routine disease surveillance and country reporting; (2) Médecins Sans Fontières (MSF), used for field data collection and as a central repository for medical data. Logs from these instances (gathered from the instances' server side) are from the period February to September 2019, and the logs together contain approximately 2.5 million requests.

DHIS2 WAPI resources (e.g., data, documents, functionality) are exposed through WAPI endpoints (e.g., `api/dataValueSets)`. Using these endpoints, consumers can access and manipulate data stored in the instance of DHIS2, data related to diseases cases (e.g., where a disease or infection spread, number or cases), organization units collecting the data (e.g., hospitals), etc. Consumers can interact with the WAPI using HTTP methods: call a GET request to retrieve a resource, a POST request to create one, PUT to update a resource, and DELETE to remove it, e.g., `GET api/charts, POST api/dataSets/ID/version`.

The second use case belongs to a WAPI offered by the Barcelona School of Informatics at the Polytechnic University of Catalonia (Facultat d'Informàtica de Barcelona, FIB, UPC). The FIB WAPI, developed by inLab research laboratory[4], provides a set of endpoints for extracting data about departments, courses, exams, rooms reservations, etc. It is a read-only WAPI, meaning that consumers can only retrieve the data using GET requests, e.g., `GET v2/assignatures, GET /v2/lectures`. It is built under REST architecture and is mainly being used by the FIB website, monitoring systems, school news screens, and several applications created for academic purposes. The log data we are using (gathered from the server side) dates from October 2019 to January 2020, and the logs contain approximately 1.8 million requests.

[3]`https://www.who.int/tools/who-integrated-data-platform`
[4]`https://inlab.fib.upc.edu/en/inlab-fib`

5.2 Data Analysis

As the data in the WAPI logs are complex (e.g., unstructured, high volume data) and somewhat noisy, the information needs to be prepared according to the requirements of process mining analysis that will be applied. Thus, we performed a pre-processing phase, which consists of four steps: (i) *data fusion* that involves the merging of log files generated from different sources, (ii) *data cleaning* which consists of removing irrelevant and noisy log entries from the files (e.g., cases containing requests with client-side errors), (iii) *data structuring* that includes user and session identification, and (iv) *data generalization* that consists of extracting general WAPI specifications from the requests in the log files [89], [43]. We have addressed extensively the problem of WAPI usage logs pre-processing in one of our previous works [43].

After the pre-processing phase, each log entry should have a distinct timestamp (to be able to establish the order of calls), a clear activity name (to identify the relevant calls made), and should refer to one case (to determine the execution of the process). Thus, a (normalized) log entry should look like this: "`24 5e93393e93c7ea99ea 2019-10-17 10:31:53 GET v2/quadrimestres/ID/assignatures 200 1233`", containing an identifier for the corresponding session (case id), application identifier, timestamp, method and endpoint (activity name), status code, and object size returned, respectively. We used the session identifiers as case ids, the request time as timestamps, the endpoints and the method as activities, and the other attributes as resources.

Considering the interaction between consumers and WAPI as a process, we applied process mining to build a metric-based process model. Simply put, for each node and edge, we calculated several figures (e.g., absolute frequency, case frequency), as specified in Section 4.

With the calculated figures for the nodes and edges and the set of patterns, we then detected the occurrence of specific patterns in our data and computed for each of them the two metrics, confidence and support.

5.3 Feedback from Programmers

In this section, we evaluate our approach by responding RQ4. We do this by providing the empirical evaluation of RQ1-3, which have been responded in the previous section.

After detecting several patterns in the usage logs of the two WAPIs in study, the next step is determining two major things: (i) the extent to which these patterns are manifesting in consumers' behavior and their needs for improvement (RQ 4.1), and

(ii) the extent to which these patterns are a real indication for change, from providers' point of view (RQ 4.2).

Table 6.1: Interviewed WAPI providers and consumers

WAPI	Developers	Experience with WAPI
DHIS2	EST, 2 developers	Consumers: Use DHIS2 WAPI to build applications
	MSF, 3 developers	Consumers: Use DHIS2 WAPI to maintain already built applications
FIB	inLAb, 1 developer	Provider: Develop and maintain FIB WAPI

For these reasons, we evaluate our approach in two directions. First, we interview DHIS2 WAPI consumers, i.e., developers that have used the WAPI and have incorporated it into their applications. With this evaluation, we aim to assess the effectiveness of our approach in understanding consumers' needs and identifying areas for improvement from the log analysis. Furthermore, we interview FIB WAPI providers, i.e., those who have designed, developed, and still maintain the WAPI's endpoints. We aim to assess from the providers' perspective whether the suggested changes are properly addressing the detected behavior. As providers are the ones equipped with the needed knowledge on the WAPIs' endpoints, resources, functionalities, and the structure of the data behind the WAPI, they can better evaluate the importance of these patterns. On the other hand, besides technical expertise, they know the environment where the WAPI is being developed and maintained, the organizations that are exposing the data/functionalities, their policies, and requirements. Thus, they can determine the feasibility and applicability of the suggested changes.

Table 6.1 provides an overview of the interviewed developers. We performed the consumers' interviews online and the provider's interviews face-to-face. The interviews followed a semi-structured format and took approximately thirty minutes per person. The patterns we detected from analyzing the usage logs were the main input for designing the interview.

5.3.1 Interviews with WAPIs consumers - DHIS2 use case

As we were able to get usage data from two consumers of DHIS2 WAPI, namely WIDP and MSF (introduced when presenting the DHIS2 case), interviewing developers from them was the most reasonable option. The two first developers were from EyeSeeTea (EST), a consulting company that maintains WIDP servers, develops, debugs, maintains, and upgrades DHIS2 official applications, as well as builds their own applications. We interviewed also three in-house developers from MSF, who are mainly working on maintaining DHIS2 applications, and not building them from scratch.

Table 6.2: Consumers' Interviews' Results

Nr.	Pattern	Dev.	Endpoint A	Endpoint B	Sup.	Conf.	Feedback
1	P2	Dev.1	GET api/programIndicators	GET api/programRules	1080	0.70	⊕
2	P5	Dev.2	GET api/userGroups	GET api/userRoles	623	0.54	⊕
3	P4	Dev.3	GET api/reportTables/ID	GET api/charts/ID	27537	0.59	⊕
4	P1	Dev.3	PUT api/events/ID/ID		125375	0.64	⊕
5	P2	Dev.4	GET api/programRules	GET api/programRuleVariables	730	0.44	⊕
6	P1	Dev.4	GET api/programIndicators		6257	0.62	⊕
7	P1	Dev.5	GET api/programIndicators		6257	0.62	⊕
8	P1	Dev.5	PUT api/events/ID		125375	0.64	⊕
9	P4	Dev.1	GET api/analytics	GET api/charts/ID	41083	0.57	⦸
10	P7	Dev.4	GET api/userSettings	GET api/attributes	10810	0.42	⊘
11	P5	Dev.5	GET api/userGroups	GET api/userRoles	623	0.54	⊘
12	P7	Dev.1	GET api/programs	GET api/schemas	838	0.07	⊗
13	P2	Dev.1	GET api/dataElementGroups/ID	GET api/categoryCombos/ID	195	0.76	⊗
14	P2	Dev.1	GET api/constants	GET api/programs	148	0.01	⊗
15	P6	Dev.2	GET api/attributes	GET api/userSettings	4057	0.27	⊗
16	P6	Dev.2	GET api/schemas/organisationUnit	GET api/attributes	2419	0.26	⊗
17	P2	Dev.2	GET api/dataElementGroups/ID	GET api/categoryCombos/ID	195	0.76	⊗
18	P1	Dev.2	POST api/dataValues		139859	0.82	⊗
19	P6	Dev.3	GET api/enrollments	GET api/events	300	0.36	⊗
20	P2	Dev.3	GET api/dataElementGroups/ID	GET api/categoryCombos/ID	195	0.76	⊗
21	P2	Dev.3	GET api/programIndicators	GET api/programs	90	0.01	⊗
22	P6	Dev.3	GET api/trackedEntities	GET api/programs	36	0.35	⊗
23	P5	Dev.3	GET api/userGroups	GET api/userRoles	623	0.54	○
24	P4	Dev.4	GET api/attributes	GET api/systemSettings	11768	0.56	○
25	P2	Dev.4	GET api/programIndicators	GET api/programRules	1080	0.70	○
26	P7	Dev.5	GET api/dataElementGroups	GET api/dataElementOperands	111	0.05	○
27	P5	Dev.5	GET api/organisationUnitLevels	GET api/dataApprovalLevels	4628	0.68	○

Before conducting the interviews, we sent to all interviewees a short questionnaire with a few more general questions regarding their background, experience working with WAPI, and a list of endpoints from the respective WAPIs that appear in one or more of the detected patterns. The interviewees were asked to select from the list the endpoints they were more familiar with, so we could customize the interviews' questions to their knowledge about the WAPIs.

We started the interview with some general questions regarding the evolution of DHIS2 WAPI, to better understand how changes are perceived from consumers' point of view (the whole set of questions can be found in 2).

In general, consumers asked for more detailed and updated documentation. They did not complain about the frequency of changes, nor the compatibility (most of the newly introduced changes are compatible with previous versions), but when even a minor change is introduced without being documented, they experience several issues with their applications. They brought the example of the `api/events` endpoint that was recently changed from version 2.37.7 to version 2.37.8, without being documented. The change affected the response: before, this endpoint used to return ten different properties, and after the change, it was returning half of them. The other properties were excluded from the WAPI response. In case developers wanted those properties to be included in the response payload, they had to specify them in the request query. As this change was not documented, developers had to notice it themselves and react afterward, so that their applications would not misbehave.

Table 6.3: Consumers' answers' categorization

Symbol	Endpoint(s) are used as in the pattern	The change is relevant (There is room for improvement)
$\oplus$	Y	Y
⃠	Y	N
$\oslash$	N	Y
$\otimes$	N	N
$\bigcirc$*	-	-

[*] Consumers do not use the endpoint(s) as in the pattern in their use cases.

Regarding the communication with providers, all the interviewed consumers confirmed that the DHIS2 community of practice is very active and that providers have made available several communication channels for consumers (e.g., JIRA, mailing lists, Slack channels). Providers respond to most of the issues raised by consumers and fix the reported bugs. This means that, to some extent, providers take into account explicit consumers' needs when implementing changes.

Next, we made more specific questions and asked the respondents about the usage of specific endpoints, and their opinion on specific possible changes that could be performed on those endpoints. We showed them endpoints from significant patterns (i.e., with high confidence and support), as well as endpoints from non-significant patterns (i.e., with low significance or support), without giving any insights regarding the value of respective metrics. This way, we could assess the importance of the detected patterns, prove that we did not miss any important issues with endpoints, and avoid a possible bias (interviewees thinking that all patterns are significant).

On average, we introduced five patterns to each developer. To evaluate as many patterns as possible, we aimed to use different examples for each developer. But, considering the set of endpoints they were more familiar with (selected from the pre-interview questionnaire), we sometimes had to show different developers the same example. However, the answers for the same patterns turned out to be neither conflictual nor contradictory.

Table 6.2 shows the results of the interviews. Table 6.3 shows the different categories where consumers' answers could fall.

Case 1: Endpoints used as in the pattern, and the change is relevant $\oplus$. In seven of the patterns (eight rows in the table as one of the pattern occurrences, nr. 6 and nr. 7, were shown to two different developers), the interviewed developers said that they use the endpoints as in the patterns and that the endpoints could be improved. In some cases, they welcomed the idea of merging the endpoints used together, and for the cases where the merging was not directly embraced (an expected behavior from consumers to not directly agree on a change, as it will be followed by changes in their applications), they admitted having some issues with those endpoints and pointed out the need for refactoring. For pattern occurrence nr. 3 (i.e., fork: `GET api/reportTables/ID` and `GET api/charts/ID)`, the suggested change to merge these endpoints together was already implemented in the latest version of the DHIS2 WAPI. The new introduced endpoint `api/visualization` unifies both `api/charts` and `api/reportTables` endpoints. As explained in DHIS2 documentation [5], the main idea behind this change was to enable consumers to have a unique and centralized endpoint that provides all types of charts and report tables as well as specific parameters and configuration for each type of visualization. As Dev.3 confirmed, this change actually simplified the WAPI, as both `api/charts` and `api/reportTables` endpoints were being used to represent the same information.

Case 2: Endpoints used as in the pattern but the change is not relevant $\oslash$. In pattern occurrences nr. 9, *Dev.1* responded that he uses the endpoints as described in the pattern, but he did not find relevant the change suggested by the pattern. This pattern had relatively high support and confidence, thus it was frequently found in the logs. The developer explained that one of the endpoints is used to extract metadata, and the other to extract data, and that is why merging them is not an option.

Case 3: Endpoints not used as in the pattern but the change is relevant

[5] `https://docs.dhis2.org/en/home.html`

⊘. In pattern occurrence nr. 10, *Dev.4* said that even though he was not using the endpoints as in the pattern (choice pattern: `GET api/userSettings` and `GET api/attributes)`, one of the endpoints (`GET api/userSettings`) could be improved. In pattern occurrence nr. 11, *Dev.5* said that, even though he does not use the endpoints as in the pattern, there is room for improving them. He admitted that the logic behind endpoint `GET api/userGroups` is too complex and not intuitive.

Case 4: Endpoints not used as in the pattern and the change not relevant ⊗. In nine of the pattern occurrences (eleven rows in the table as one of the pattern occurrences, nr. 13, nr. 17, and nr. 20, were shown to three different developers), the interviewed developers responded that they do not use the endpoints as in the patterns, and the suggested change was not relevant. In seven of the cases, the patterns had low support and low confidence, so their answers were the expected ones. But two of the patterns that got this answer had high confidence. We showed the direct follow pattern (`GET api/dataElementGroups/ID, GET api/-categoryCombos/ID`) with confidence = 0.76, to three different developers and got the same answer: they do not use these endpoints one after the other, and their merging was not seen as beneficial from their point of view.

The other pattern occurrence with high confidence, and at the same time high support, that got the same answer was the reflexive loop of the endpoint `POST api/dataValues` (row nr. 18). The developer said that the endpoint was working fine, they did not have any issue with it, thus there was no room for improvement. Interestingly, this pattern (reflexive loop) is a case where some consumers may not experience any inconveniences with the endpoint (if network latency, i.e., ping time is low), thus no improvement by the change. They may call the endpoint in a loop, and each iteration may generate a WAPI request. In high latency situations, the delay coming from too many requests might be significant compared to one large request. Differently, *Dev.4* and *5* showed interest and approved our suggestion to change the endpoint `GET api/programIndicators` (reflexive loop, nr. 6 and nr. 7), so that they could call it one time to get the data they need, instead of making several consecutive calls.

Case 5: Endpoints not used as in the pattern in their use cases ◯. In five of the pattern occurrences, the interviewed developers said that in their use cases, they do not use the endpoints as in the patterns, but they could not say anything regarding the suggested change, as long as their usage of the endpoints was limited in few use cases. While one of the pattern occurrences had low support and confidence (nr. 26), the other four had relatively high support and high confidence. We happen to introduce two of these patterns (nr. 2, nr. 11, and nr. 23 `GET api/userGroups` and `GET api/userRoles`, and nr. 1 and nr. 25 `GET api/programIndicators` and `GET api/programRules`) to three different developers. For the first pattern occurrence (`GET api/userGroups` and `GET api/userRoles`) *Dev.2* and *5* said that the logic behind `GET api/userGroups` could be improved, and that it needs some refactorings. For the second occurrence (`GET api/programIndicators` and `GET api/programRules`), *Dev.1* said that as long as there are cases where he uses these endpoints together, changes could be applied to improve them. These are the only cases where the same pattern, got different (but not contradictory) answers from different developers.

5.3.2 Interviews with WAPIs providers - FIB use case

As already introduced in Section 5, FIB WAPI is a private WAPI, built to mainly serve the creation of the faculty web page and other related applications. Therefore, providers and consumers happen to be from the same development group. Actually, this is not an isolated case. Typically in banks, other financial institutions, or several companies, the development group creates a private (W)API to specifically serve their needs for the companies/institutions' applications. Thus, providers' decisions in applying changes in the WAPI will be directly affected by consumers' needs, expressed explicitly.

Table 6.4: Provider's Interview Results

Nr.	Pattern	Endpoint A	Endpoint B	Sup.	Conf.	Feedback
1	P2	GET v2/assignatures/ID/guia	GET v2/competencies	68942	0.74	⊕
2	P1	GET v2/competencies/categories		33668	0.93	⊕
3	P2	GET v2/sales	GET v2/sales/reserves	157647	0.71	⊗
4	P4	GET v2/quadrimestres/actual	GET v2/jo/assignatures	6013	0.69	⊗
5	P6	GET v2/assignatures/requisits	GET v2/competencies	149	0.07	⊗
6	P7	GET v2/jo/avisos	GET v2/noticies	497	0.01	⊗

We interviewed the main developer of FIB WAPI (the whole set of questions can be found in Appendix A He stated that in most cases, consumers' needs to create, change and maintain their applications were the starting point of a change in WAPI. When they need new data, not supported by the current endpoints of the WAPI, they create the respective endpoints for that data.

Table 6.4 shows the results of the interview. Table 6.5 shows the different cate-

Table 6.5: Providers' answers' categorization

Symbol	The change is relevant	The change is feasible
$\oplus$	Y	Y
$\oslash$	Y	N
$\otimes$	N	-

gories where providers' answers could fall. The answers' categories in Table 6.5 are different from the ones in Table 6.3 because, from the providers' side instead of talking about the usage of endpoints (providers design, build, maintain, and change the endpoints), we talk about the relevance of a suggested change, and its implementation feasibility.

The change is relevant and feasible $\oplus$. The developer showed interest in two of the introduced patterns' occurrences, both of which had high support and confidence. He explained the reasons behind the decision to design the endpoints in the current way. Even though he agreed that the endpoints could be improved by merging them or by adding attributes, he had decided to create the endpoints in that way because of their specific use in the applications.

The change is relevant, but it is not feasible $\oslash$. In our case, we did not get any response for this category. Providers may admit that the suggested change is relevant, but for other reasons (that cannot be observed in the logs e.g., security, privacy) the change may not be feasible and they cannot implement it.

The change is not relevant $\otimes$. The developer did not show interest in four of the patterns' occurrences. While two of them had low support and confidence (nr.5 and nr.6), the two others (nr.3 and nr.4) had high values for these metrics. While endpoints GET v2/quadrimestres/actual and GET v2/jo/assignatures (pattern occurrence nr.4) need different authorization level to be accessed (GET v2/jo/assignatures returns the subjects where a specific logged-in students is enrolled), endpoints GET v2/sales and GET v2/sales/reserves (pattern occurrence nr.3) should not be merged as their functions are well defined this way.

6 Discussion

In this section, we discuss interviews' answers and interpret the main findings. In the end, we summarize the threats to validity.

We were able to get very useful insight from the answers the consumers gave to the general questions of the interviews. While stressing the importance of updated documentation, developers admitted that they would even prefer the WAPI to offer several ways of doing the same task, as long as all the endpoints are well explained in the documentation. While this may give the impression of creating confusion or increasing the complexity of the WAPI, detailed documentation can mitigate this adverse effect, and even turn it into an advantage by increasing the flexibility for consumers. This feedback can be very valuable for providers when they decide on the changes they will implement on the WAPI. Some changes may cause an adverse effect on consumers (e.g., breaking their applications, increasing the complexity of WAPI). Knowing how to mitigate these effects, providers may be more relaxed when implementing the changes.

Furthermore, developers confirmed the availability of different communication channels between providers and consumers. Providers usually reacted to issues raised by consumers, making these channels very conducive. This means that, to some extent, providers take into account consumers' needs when implementing changes. Our approach will help them to apply a more in-depth analysis of consumers' needs, by gathering their feedback in a more straightforward, inexpensive, and scalable way.

From all the interviewees' answers to the specific questions, which are detailed in previous section, the ones that may be of more interest to be discussed here are those that do not accept the changes implied by patterns with high confidence and support and those that accept changes implied by patterns with low confidence and support. As we cannot arbitrarily decide on thresholds' values for the confidence and support (it may depend on how clean the logs are, or how complex the WAPI is) to later classify the patterns as significant or unsignificant (and categorize them as false positive/negative or true positive/negative), we make a more qualitative discussion and generally review the most controversial answers.

Patterns that have high confidence ($\geqslant$ 0.5) and support, whose suggested changes are not welcomed by consumers or providers. Patterns fall in this category due to the presence of applications' end users' behavior in the WAPI logs. The way consumers' applications are built and the way their users interact with them, may create some frequent patterns in the WAPI usage logs, that do not indicate the need for change. For instance, if the users of an application access two features in a certain order, the WAPI endpoints incorporated in those features will appear in the logs as called together. This was the case with one of the patterns we showed to consumer developers. The pattern occurrence appeared with high confidence of 0.76, and relatively low (considering the margin of error due to the noise in the logs, and the size of the log file) support of 195. The developer confirmed he was not using together the endpoints (i.e., direct follow: `GET api/dataElementGroups/ID` and `GET api/categoryCombos/ID`). However, when he demonstrated us the consuming application, the features (i.e., tabs) using these endpoints were next to each other in the application interface, presumably guiding the end-users to access them one after the other. Thus, the behavior of the application's end-user was reflected in the logs.

Second, even though patterns may have high metrics' values, semantically they cannot be merged. This was the case of the endpoints `GET api/analytics` and `GET api/charts/ID`. The fork pattern appeared with high support (41083) and relatively high confidence (0.57). However, one of the endpoints was used to get metadata, and the other to get data. Thus, their merging was off the table. This example is very interesting, as it implies the need to semantically check the patterns during pattern detection. Providers may exclude beforehand those patterns whose respective endpoints are built to extract different kinds of information (e.g., metadata and data).

The unavoidable presence of this group of patterns in the analysis results makes it necessary the evaluation of the patterns from the WAPI providers, that possess the needed domain knowledge to filter out the insignificant patterns. The blind evaluation based only on the value of metrics may result in wrong conclusions.

Patterns that have low confidence and/or support, but indicate the need for changes in the WAPI. Patterns can fall in this category if the dataset is not large enough. As one of the main aspects of our approach is the repetition of consumers' behavior, the lack of data may impede the discovery of interesting patterns. When showing non-significant patterns, developers did not show interest in any of them. They gave examples of using the endpoints independently of each other, thus not agreeing with the changes to merge the endpoints.

Threats to validity: Most of the internal validity threats concerned the pre-processing of the usage logs. As we mentioned in Section 5.2, the pre-processing of the usage logs was not trivial. We tackled some challenges when introducing the main steps of pre-processing. Each challenge can affect the validity of the results if the related issues are not identified and properly handled. We can mention here the session identification challenge. Most of the WAPIs are stateless, meaning that the server does not store the state, thus no sessions are generated. We constructed the sessions by assigning identifiers based on the 30 minutes timeout approximation (requests with a time difference of fewer than 30 minutes, coming from the same user, are part of the same session). This might add noisy sequences of calls (for applications with smaller timeout values) or exclude important ones (for applications with greater timeout values). To mitigate the risk of the 30-minute approximation affecting the analysis, we computed several metrics for each pattern and the respective endpoints, so that the providers can have a more complete view of their usage.

One of the construct validity threats consists of misinterpreting the behavior of applications' end-users as applications' behavior toward the WAPI. The way end-users interact with the applications should not interfere with the analysis of the way applications consume the WAPI. To mitigate this threat (and to assess the effectiveness of our approach), we included in the loop the WAPI providers to properly evaluate the significance of the detected patterns.

Regarding internal validity, being aware of the possible researcher biases that might impact the interviewees' answers, we tried to be as neutral as possible during the interviews, paying particular attention to the questions' creation and the patterns' occurrences we introduced to the interviewees. Besides randomly selecting patterns' occurrences (taking into account the endpoints selected by interviewees in the questionnaire) with very diverse metrics' values, we chose not to show these values to the interviewees. Thus, their answers would not be influenced by the frequencies of the patterns, but only by their knowledge and experience with the respective WAPIs.

Furthermore, we were aware of the threat to validity concerning the subjective opinions interviewees may have regarding several endpoints. We mitigated it by showing to different interviewees same patterns and then observing their feedback. In none of the cases, their feedback was conflictual, or contradictory, affirming that they have assessed each pattern objectively based on their experience using the WAPIs.

To better understand the needs of WAPI consumers, one may have to include in the analysis the activities that consumers are performing after accessing the WAPI. As the WAPI is not used in isolation, its activities and the activities in the consumers' applications, outside the WAPI, are correlated. For instance, after accessing some data from the WAPI, an application may parse the data or cast the data type, and then use them according to its functionality. In order for the WAPI providers to identify the need for changing the data type, or shaping the data in a more useful way for their consumers, they should correlate and analyze both the WAPI usage logs and applications running logs. But, knowing the independent nature of WAPIs, how loosely coupled they are with their consumers [70] and taking into account that consumers' applications code is not always public, analyzing these services together becomes unrealistic. As we analyze only the WAPI usage logs, our challenge is to understand consumers' behavior and identify their needs by making use of their footprints in the logs.

7 Related Work

In this section, we review those works that are related to the intention behind (W)API evolution and the most followed practices by providers, as well as the use of process mining in analyzing WAPI behavioral patterns.

Granli et al. [32] analyzed the driving forces of traditional API evolution using the existing software evolution theories, more specifically Lehman's laws. They showed that the two largest forces driving API evolution were the need for new functionalities and usability improvements. Their study revealed that business requirements and API consumers stand behind these forces, but they did not further elaborate on these factors. Sohan et al. [82] investigated changes made when new WAPI versions were released, and how these changes were documented and communicated to consumers, without considering the reasons that lead providers in applying those changes. Several works are done in classifying the changes that often happen to WAPIs from older to newer versions [41], [97], [53]. The addition of new elements [41], [97] and refactorings [53] were the most common ones. Li et al. [53] analyzed the API consumers' reaction toward the changes; however, they did not examine how the need for these changes was manifested in the consumers' behavior.

Regarding providers' practices when evolving their WAPIs, Espinha et al. [26] investigated four well-known WAPIs (Google Maps, Facebook, Twitter, Netflix) and the evolution policies their providers follow. Their study showed that there is a lack of industry-standard regarding the evolution of WAPIs, and different providers follow different practices. However, the authors were more focused on the way providers introduce the new changes and communicate them to consumers, rather than on the origin of the changes. Lamothe et al. [46] conducted a systematic review of API evolution literature. Besides the continuing change and growth, the increasing complexity and declining quality, and the conservation of familiarity (all considered worthy-challenged by the authors), they particularly stressed the need to leverage the feedback systems involved in API evolution. Mathijssen et al. [60] performed a systematic literature review on different API management practices applied by providers. Access logging and usage monitoring were commonly followed by API providers, but with the aim of providing performance statistics or traffic metrics. Medjaoui et al. [62] accentuate the need for proper and continuous API Management, by proposing guidelines to build, deploy, operationalize, refine, and evolve (W)APIs continually at scale.

There are few studies that consider consumers' behavior as feedback that should be taken into account when evolving a WAPI. Zibran et al. [105] analyzed the issues raised by API consumers in five different bug repositories to identify the most reported API usability issues. Their study involved consumers, but not by analyzing the way they used the API, but the issues reported explicitly to the providers. Macvean et al. [56] analyzed as well the usability of WAPIs. They took data from Google API Explorer to identify WAPIs, with which developers struggle more and spend more time and effort in learning and using. They were focused on the way developers perceive and learn the WAPI, and not how they were using endpoints based on the functionality. On the same line, is one of our previous works [42], where we analyzed development logs to measure the usability of WAPIs, to further suggest changes that would increase the perceived usability from consumers' point of view. Suter et al. [87] analyzed WAPIs usage logs using trained classifiers in order to infer WAPI description. Even though with a goal different from ours, they pointed out the incomplete and noisy nature of WAPI usage data. They encourage future work not only on mitigating the quality issues of these data but also on broadening the goals spectrum of the usage logs analysis.

Related to the use of process mining in analyzing WAPI usage logs, we can find several works akin to our study in some particular aspects, like input data where the process mining is applied, not focusing on the end-to-end process but on part of it, etc.

Aalst [91], [93] showed the potential of applicability of process mining in the services domain, identifying as well the main challenges like the correlation of the events. In both works, it was pointed out the need for additional research in improving the applicability of process mining in loosely coupled systems.

Gunther and Aalst [34] focused on addressing the problem of the less structured processes (like accessing WAPI), where the discovered models are usually "spaghetti-like", not understandable, and uninformative. They presented a new process mining approach to overcome this problem by simplifying the model using two metrics: significance and correlation. Even though they state that their approach is universally applicable, we do not opt for any simplifications as we are interested in the direct order of WAPI calls and simplifications may affect the accuracy of patterns.

Zhong et al. [103] have developed a framework for automatically mining API usage patterns (MAPO). Different from us, they mine applications' code snippets (no usage logs), focus on traditional API (no web API), and recommend usage patterns for consumers to better use the API (no for providers to evolve the API based on the implied changes). Wang et al. [96] claim to outperform MAPO by introducing UP-Miner approach. They introduce two quality metrics, succinctness and coverage, that help to mine better API patterns (not redundant, more practical, more qualitative), so that developers (API consumers) would adopt them more easily. The same goal drove Nguyen et al. [67]. The authors introduce a graph-based statistical language that analyzes applications' source code to suggest accurate API usage patterns. However, same as [103] and [96], they focus on traditional API and mine applications' code.

Leemans et al. [48] introduced a technique to discover frequent episodes (i.e., partially ordered set of events) in event logs, using the notion of frequent itemsets and association rules. Their work, like ours, is somehow positioned between process mining and pattern mining, but unlike us, they do not pre-define any patterns but look for the most frequent ones. Bose and Aalst [38] focuses on discovering common patterns in process execution traces. They pre-define a few patterns that capture the manifestation of commonly used process construct and propose an approach to form an abstraction of activities using these patterns. Even though the presented patterns could be applied in several domains, we had to define our own set of patterns, which are not only specific to the WAPI case but also capture specific consumers' behavior and imply WAPI change.

Even though the above-mentioned works differ from ours regarding the input of the approach, the goal, or the methods followed, they helped us identify the current gaps in WAPI evolution practices. Reviewing those works helped us gain the right perspective on WAPI evolution and the application of process mining in our use cases. We identify a set of usage patterns, by analyzing usage logs, thus avoiding the analysis of applications' source code (not always available). We associated each pattern with potential changes in WAPI and presented their detection in the logs, opening new opportunities for providers to differently evolve their WAPIs (driven by consumers' behavior).

8 Conclusion and Future Work

In this paper, we presented an approach to how the analysis of WAPI usage logs can help providers to better understand the consumers' needs for specific improvements. Typically, WAPI providers log all the requests against the WAPI. Consumers' behavior is recorded on these usage log files. Hence, providers can infer useful information from their analysis. We defined seven behavioral patterns, whose occurrence can indicate the need for potential changes on the WAPI. We observed the occurrence of these patterns in two real-world examples and suggested potential changes that could improve the WAPI from the consumers' perspective. The feedback from consumers and providers of the WAPIs in the study confirmed the promising potential of our approach. Moreover, recently, DHIS2 WAPI providers had been introducing a few changes, motivated by the same ideas and goal as in our approach: to enable consumers in having well-defined and simplified endpoints, that would ease their experience with the WAPI.

In our future work, we aim to define more patterns, that cover more usage scenarios and capture more complex consumer behavior. In order to evaluate the effectiveness of the proposed patterns and changes for the consumers, we plan to analyze the WAPI usage logs after the implementation of these changes, to see if the consumers are using the WAPI in the way we thought they will. Moreover, we intend to focus on the creation of useful and relevant migration scripts for the changes we suggest.

We plan to perform an additional evaluation to assess the providers' benefits coming from applying our approach during their WAPI evolution. Besides evaluating the benefits, a very interesting direction to expand the work would be to analyze and evaluate the providers' maintainability burden that specific changes may bring. Studying the break-even point between consumers' retention and satisfaction and the maintenance burden caused by the addition of some functionality would be very promising, as currently, it represents a research gap. Another way this work can be expanded is by studying the WAPI evolution from the business point of view: analyzing the companies' requirements that drive several changes. This way, the picture of the intent behind the changes will be more complete, and the prioritization of the changes coming from both consumers and companies can be somehow automated.

Chapter 7

PatternLens: Inferring Evolutive Patterns from Web API Usage Logs

The paper has been published as a demo paper in the *Intelligent Information Systems: CAiSE Forum 2021.*
DOI: http://dx.doi.org/10.1007/978-3-030-79108-7_17

Abstract

The use of web Application Programming Interfaces (WAPIs) has experienced a boost in recent years. Developers (i.e., WAPI consumers) are continuously relying on third-party WAPIs to incorporate certain features into their applications. Consequently, WAPI evolution becomes more challenging in terms of the service provided according to consumers' needs. When deciding on which changes to perform, besides several dynamic business requirements (from the organization whose data are exposed), WAPI providers should take into account the way consumers use the WAPI. While consumers may report various bugs or may request new endpoints, their feedback may be partial and biased (based on the specific endpoints they use). Alternatively, WAPI providers could exploit the interaction between consumers and WAPIs, which is recorded in the WAPI usage logs, generated while consumers access the WAPI. In this direction, this paper presents

PatternLens, a tool with the aim of supporting providers in planning the changes by analyzing WAPI usage logs. With the use of process mining techniques, this tool infers from the logs a set of usage patterns (e.g., endpoints that are frequently called one after the other), whose occurrences imply the need for potential changes (e.g., merging the two endpoints). The WAPI providers can accept or reject the suggested patterns, which will be displayed together with informative metrics. These metrics will help providers in the decision-making, by giving them information about the consequences of accepting/rejecting the suggestions.

1 Introduction

As the use of web Application Programming Interfaces (WAPIs) is increasingly growing, their evolution becomes more challenging in terms of the service provided according to consumers' needs. While a lot of effort is put into analyzing consumers' struggles when WAPIs change [27, 53, 97], little is known about providers' burdens: what, how, and when to evolve [3]. When deciding on which changes to perform, besides several business requirements (from the organization whose data are exposed), providers should take into account the way consumers use the WAPI. Moreover, knowing the impact that changes usually cause to the consumers, they have to strike a balance between not imposing irrelevant, unexpected, frequent changes and providing an up-to-date, maintainable, bug-free WAPI, that fulfills their needs [41].

While consumers may report various bugs or may request new endpoints (i.e., URLs to access WAPIs resources), their feedback may be partial and biased (based on the specific endpoints they use), as well as difficult to gather and interpret at scale [102]. Alternatively, WAPI providers could exploit the interaction between consumers and WAPIs, which is recorded in the WAPI usage logs, generated while consumers access the WAPI. Every time a consumer application makes a request, a log entry is generated and stored in the usage log file. Therefore, consumers' behavior is recorded in these logs, and their analysis can obliquely reveal consumers' needs for new features hidden under several workarounds (solutions found by WAPI users that allow them to get data, functionality, or features they need, but that are not yet implemented by providers), or find room for potential improvements.

As applications are the actual WAPI consumers, we should consider the different ways they consume WAPIs over their own lifecycles. Basically, applications interact with the WAPIs during design time and runtime, over both of which manifest different aspects of their behavior. Following from this, we distinguish two types of logs: (i) development logs, generated while developers build and test their applications, and

(ii) production logs, generated while applications are being used by end users, meaning that the WAPI requests are predetermined by the implemented functionalities of the applications. We make this distinction as each of these log types, even though provide useful information about WAPI consumption, can be analyzed in different ways. But how much are these logs used and analyzed, beyond the typical traffic monitoring? Several works [42, 56] suggest analyzing development logs to measure the usability of WAPIs and thus perform changes that increase the perceived usability from consumers' point of view. In addition, there are various WAPI monitoring tools available that take as input the WAPI usage logs, but they are mostly oriented toward providing reporting dashboards or automatic alerting in case of WAPI failure [23]. Providers have all this potentially insightful, large volume of data that is being generated, but not enough proactively used for evolution.

To address the problem of understanding consumers' needs, we present PatternLens, a tool that aims to support providers in planning the changes by analyzing consumers' behavior recorded in the WAPI usage logs. We make use of process mining techniques and compute from the logs a set of metrics regarding the real consumption of WAPI endpoints by consumers, like the calling frequency of the endpoints, the frequency of a sequence of endpoints, etc. Using these metrics and a pre-defined set of patterns, we detect from the logs all the patterns, whose occurrences imply the need for potential changes (e.g., merging two endpoints called always one after the other). The WAPI providers can accept or reject the suggested patterns, which will be displayed together with informative metrics, helping providers in the decision-making.

The remainder of the paper is organized as follows. Section 2 presents the basic concepts, derived from process mining techniques, used and implemented in the tool. Section 3 introduces the tool, its main features, and its functionality. Section 4 reports preliminary experimental results of the tool's usability and ease of use evaluation, while Section 5 presents the future plans for extending the tool.

2 Background: Process Mining in the WAPI context

Process mining is a process-oriented data mining discipline that uses event logs to extract process-related knowledge. We give the definition of some fundamental concepts of process mining, as presented by Van der Aalst [92], adapt them to the WAPI domain, and use them to give the definition of concepts referred to in the rest of the paper:

- *Activity* - a specific step in the process. For WAPIs, activities are calls to the endpoints, i.e., the WAPI resources URL.

- *Case* - a process instance. We refer as a case to a set of requests that an application is submitting to the WAPI during a certain time period, commonly referred as session.
- *Event* - an activity occurrence. An event refers to an activity and belongs to a specific case. We refer to WAPI requests as events and identify them by activity names (request methods like GET, POST, and the endpoint).
- *Event log* - a set of cases. A WAPI usage log contains all the requests (i.e., events) that consumers (i.e., applications) make against the WAPI.

Using these concepts, we define a sequence as an ordered occurrence of events within a case, and a pattern as a frequent sequence that when fulfilling some pre-defined conditions, indicates the need for a specific change.

In order to apply process mining, an event log should have at least three attributes: (i) case identifier that identifies the case to whom each event belongs, (ii) activity name that identifies each event, and (iii) timestamp that indicates the time when an event occurs. The presence of these three attributes allows inferring process insights from the event log. Certainly, other attributes that might store additional information about the events (e.g., device IP, application ID, status code), whenever available, add value to the analysis.

In the WAPI context, process discovery consists of creating a graph-based process model from the event log, where nodes represent the endpoints being called, and edges the calling sequence of two endpoints. Even though we do not visualize the process model, we aim to gather statistical information about it (e.g., the frequency of calling one endpoint after another).

3 PatternLens overview

PatternLens takes as input the usage log file (i.e., event log) containing all the WAPI calls from several consumers' applications. We assume that the logs are already cleaned and prepared, meaning that (i) every event (i.e., request) belongs to one case (i.e., session), (ii) every event has a distinct timestamp, making able to build the right sequence of calls, and (iii) the activities' names are precise enough to identify each process step, so that no two different process steps to appear with the same activity name, and at the same time no the same process step to appear with different activity names.

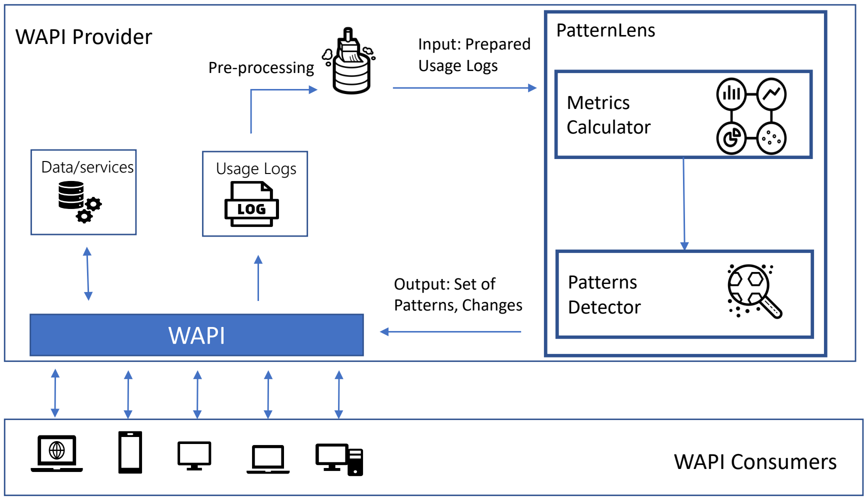


Fig. 7.1: PatternLens in the full context of WAPI - consumers interaction

Based on process mining concepts, the tool first computes a set of metrics for each WAPI endpoint and sequence of endpoints as appearing in the file. Then, using a pre-defined set of patterns and the computed metrics, it detects and displays to the users all the patterns, of different types. Along each pattern, the users are presented with the suggested changes that the patterns imply, as well as various metrics that might help them in selecting the patterns that better fit their interests. The users are given the possibility to accept the patterns that are interesting to them (i.e., the implied changes are feasible from the providers' point of view and would really improve the WAPI with regard to consumers' needs), reject the ones that do not seem interesting, or ignore the ones for whom they are neutral or do not have any conscious thought. We want to note that PatternLens is designed to be used by WAPI providers in analyzing consumers' behavior. As such, we assume that its users (i.e., providers) have a strong prior knowledge of the WAPI, needed in order to properly accept/reject patterns and the recommended changes. In the following, we introduce and explain two main parts of the tool: the metrics calculator and the patterns detector (see Figure 7.1).

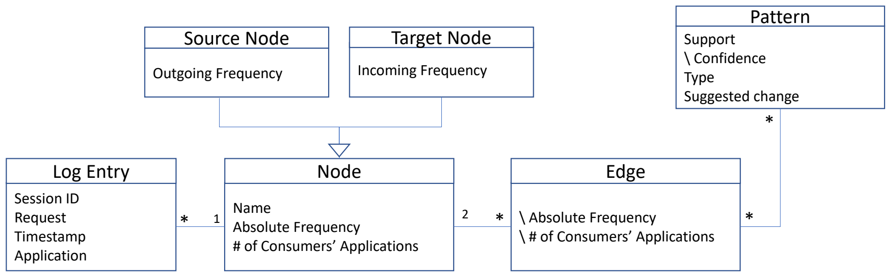


Fig. 7.2: PatternLens: Class Diagram

3.1 Metrics calculator

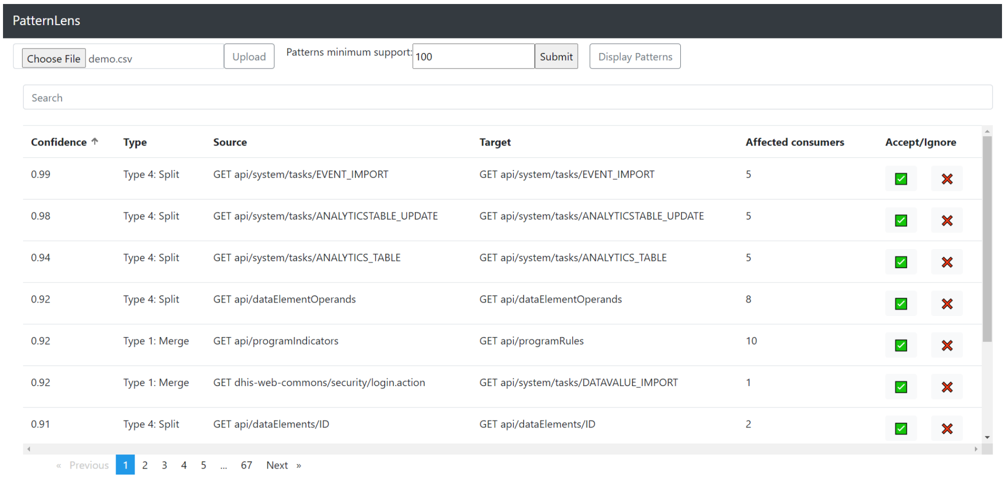
PatternLens considers the interaction between consumers and WAPI as a process. As such, to build the process model, we refer to the process activities (calling method and endpoint) as nodes, and to the order two endpoints are being called by the consumers, as edges. Before detecting the patterns, PatternLens defines all the nodes and edges, by extracting for each of them a set of attributes and computing some metrics (Figure 7.2).

For the nodes we compute and make use of the following metrics:

- *Absolute frequency:* the total number of times that an endpoint was called.
- *Incoming frequency* (for target nodes): the total number of times that an endpoint was called after other endpoints.
- *Outgoing frequency* (for source nodes): the total number of times that an endpoint was called before other endpoints.
- *# Consumers' application:* the number of applications that have called the endpoint in the study.

Each edge combines two nodes: a source node (i.e., the first called endpoint in a two-node sequence), and a target node (the second called endpoint in a two-node sequence). We define the edges by the following derived metrics:

- *Absolute frequency:* the total number of times that a sequence of endpoints was followed.
- *# Consumers' application:* the number of applications that have called the sequence of endpoints in the study.


PatternLens

Choose File demo.csv Upload Patterns minimum support: 100 Submit Display Patterns

Search

Confidence ↑	Type	Source	Target	Affected consumers	Accept/Ignore
0.99	Type 4: Split	GET api/system/tasks/EVENT_IMPORT	GET api/system/tasks/EVENT_IMPORT	5	✓ ✗
0.98	Type 4: Split	GET api/system/tasks/ANALYTICSTABLE_UPDATE	GET api/system/tasks/ANALYTICSTABLE_UPDATE	5	✓ ✗
0.94	Type 4: Split	GET api/system/tasks/ANALYTICS_TABLE	GET api/system/tasks/ANALYTICS_TABLE	5	✓ ✗
0.92	Type 4: Split	GET api/dataElementOperands	GET api/dataElementOperands	8	✓ ✗
0.92	Type 1: Merge	GET api/programIndicators	GET api/programRules	10	✓ ✗
0.92	Type 1: Merge	GET dhis-web-commons/security/login.action	GET api/system/tasks/DATAVALUE_IMPORT	1	✓ ✗
0.91	Type 4: Split	GET api/dataElements/ID	GET api/dataElements/ID	2	✓ ✗

« Previous 1 2 3 4 5 ... 67 Next »

Fig. 7.3: PatternLens: Application Interface

3.2 Patterns detector

Instead of visualizing the process map with all the sequences of calls, we filter only those parts of the process, where we see a specific behavior of the way consumers use the WAPI. We pre-define the set of patterns, whose occurrence may imply the need for potential changes, rather than extracting all the frequent patterns from the model to redundantly display them to WAPI providers without giving any hints on the behavior these patterns manifest.

To measure the effectiveness of the patterns, we use two basic concepts from association rules, namely the support and confidence metrics, and adapt them in the context of our approach. We refer as *support* to the number of times a pattern appears in the log file. We give the users the possibility to enter the desired minimum support, based on the size of the file they upload and how specific or general they want the patterns to be. We refer as *confidence* to the relative frequency of the patterns, regarding the source and the target node. As such, if a pattern has high support (i.e., high absolute frequency: it appears too often in the log file), but this frequency is too low compared with the absolute frequency of the source and the target nodes, then the pattern will have low confidence. The tool presents the patterns to the users in a tabular form as in Figure 7.3.

We add to each pattern the metrics related to the pattern or the source/target nodes (e.g., confidence, absolute frequency, number of applications where the pattern appears), to better inform the users about the occurrence of the pattern and help them

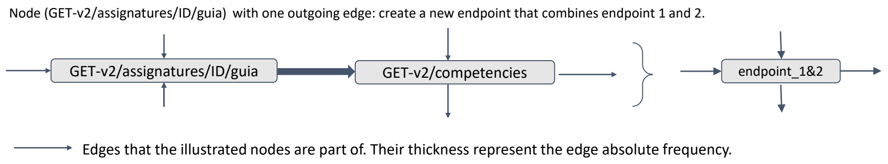
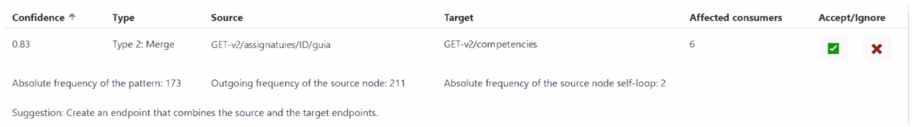

Node (GET-v2/assignatures/ID/guia) with one outgoing edge: create a new endpoint that combines endpoint 1 and 2.
GET-v2/assignatures/ID/guia
GET-v2/competencies
endpoint_1&2
Edges that the illustrated nodes are part of. Their thickness represent the edge absolute frequency.

Fig. 7.4: The pattern and the change it implies



Confidence ↑	Type	Source	Target	Affected consumers	Accept/Ignore
0.83	Type 2: Merge	GET-v2/assignatures/ID/guia	GET-v2/competencies	6	✓ ✗

Absolute frequency of the pattern: 173 Outgoing frequency of the source node: 211 Absolute frequency of the source node self-loop: 2

Suggestion: Create an endpoint that combines the source and the target endpoints.

Fig. 7.5: The pattern as displayed in PatternLens

better understand consumers' behavior. Along with the patterns, we give a short description of the implied changes (e.g., merging the two endpoints, and creating a new attribute).

Figure 7.4 gives an example of a pattern that the tool is able to detect. Figure 7.5 depicts how PatternLens displays the pattern. As seen from Figure 7.4, the GET-v2/assignatures/ID/guia endpoint, responsible to get data about specific courses in the university (source node), has one outgoing edge, the one toward GET-v2/competencies, responsible to get data about the required competencies to take a course (target node). This means that after getting information about a specific subject, consumers always call the endpoint for the competencies of the course. For this reason, we suggest merging these two endpoints in one, which combines the data from both of them, reducing this way the number of calls consumers have to submit to get the desired data. PatternLens is able to detect this type of pattern and displays it as in Figure 7.5. Along the pattern, that is defined by the source and target node, the user is informed about the confidence of the pattern ($confidence = \frac{edge\_abs\_freq}{out\_freq_s - self\_freq_s} = \frac{173}{211-2} = 0.83$), the pattern absolute frequency, the outgoing frequency of the source node, the frequency of the source node self-loop, and the change implied by the pattern. The shown metrics are different for every type of pattern, as the tool presents the most relevant ones depending on the way the pattern is being detected.

4 Onsite demonstration

In the onsite demonstration, we will present the functionality of PatternLens using log files from two real-world examples of different domains, namely health, and education. The first log file comes from the District Health Information Software 2 (DHIS2) WAPI. DHIS2 is an open-source, web-based health management information system platform used worldwide by various institutions and NGOs for data entry, data quality checks, and reporting. It has an open REST WAPI, used by more than 60 native applications. The second log file belongs to WAPI of the Barcelona School of Informatics at the Polytechnic University of Catalonia (Facultat d'Informàtica de Barcelona, FIB, UPC). It is built under REST architecture and is mainly being used by the FIB website, monitoring systems, school news' screens, and several applications created for academic purposes. The API provides a set of endpoints for extracting data about departments, courses, exams, room reservations, etc. With these two examples, we aim to show that PatternLens performs the same, despite the domain of the WAPI.

Currently, PatternLens is able to detect four different pattern types. As we assume that the users of the tool should have prior knowledge of the WAPI to decide on accepting or rejecting a pattern, and as the demo participants are not expected to be familiar with our examples, we will encourage the participants to evaluate the usability of the tool. Nevertheless, all the elements of the tool and the examples will be well-explained, so that non-WAPI experts can assess the potential of PatternLens in supporting providers in planning WAPI evolution.

5 Future work

We aim to further extend the PatternLens features in the following directions:

- Provide the users with more types of patterns.
- Quantify the impact that the suggested changes may cause if the providers decide to implement them. We plan to measure the effectiveness of the changes by predicting the excepted behavior of the consumers.
- Improve the tool according to historic data about users' selections. Currently, PatternLens stores information about the metrics of the patterns that users select or reject. We aim to improve the way the information is presented to them, based on their feedback.

Chapter 8

Conclusions and Future Directions

1 Conclusions

This work presents our approach for analyzing web API usage logs for evolution purposes. The main objective of this thesis is to provide a new framework to help providers better plan and carry out the evolution of their web APIs. We advocate for the important role of consumers' behavior and their interaction with web APIs in web API evolution planning, meaning that the way consumers use the web APIs should be one of the main factors impacting providers' decisions. As such, providers should not only consider consumers' requirements, but they should also better gather and analyze their feedback so that the new changes fit consumers' expectations and needs. We summarize the most important findings of this work, structuring them according to Chapters 2 to 7. In the end, we introduce different directions on how this work can be further extended or improved.

Chapter 2 provided an overview of web API evolution from both the consumers' and providers' perspectives. We identified and classified the most common changes that occur during the evolution process. Furthermore, we delved into the artifacts used by providers and consumers when evolving or upgrading web APIs, to better understand how providers carry out this process (e.g., announcement, documentation, explaining the rationale behind the changes) and how consumers undergo web API evolution (e.g., adapt to the changes, migrate to the new endpoints). The study showed that the majority of the changes (more than 75% of them) were non-breaking changes, and a few of them (less than 50%) were documented in at least one of the used artifacts. Thus, consumers tend to postpone the upgrade until the providers disconnect the old versions. The results of this work regarding the classification of the changes and their handling, inform the subsequent steps of our approach.

Chapter 3 focused on the change-proneness of web API and the role of consumers' usage (extracted from the usage logs) in properly predicting it. We constructed models based on design metrics, usage metrics, and design and usage metrics combined. The addition of usage metrics in the prediction not only improved the accuracy of the results but also enabled early prediction of changes that providers implement at later stages.

Chapter 4 focused on web API usage logs, highlighting their potential beyond traffic monitoring. Based on consumers' application lifecycle, we identified two types of usage logs, naming development and production logs, for each of which we introduced a set of analyses that can be applied to them for web API evolution purposes. However, we also noted the challenges associated with using these logs, such as the need for tedious pre-processing due to their unstructured and voluminous nature, and the lack of standardization by providers. We provided recommendations for mitigating these challenges and suggested ways to improve data logging.

Chapters 5 and 6 introduced the two main branches of our approach. Chapter 5 covered the analysis of development logs with regard to web API usability. Specifically, we adopted a usability taxonomy in our web API case and characterized all attributes and sub-attributes of the taxonomy in indicators, most of which could be traced in the web API usage logs. We saw that the classification model built with the help of the metrics we defined to quantify the indicators, was able to detect endpoints with poor usability.

Chapter 6 covered the analysis of the production logs. We defined a set of seven behavioral patterns, whose occurrence could indicate the need for potential changes on the WAPI. We detected the suggested patterns in two different web API usage logs and evaluated the significance of the occurrences of the patterns and the changes they suggest by interviewing their providers and customers. Their feedback confirmed the promising potential of our approach.

Chapter 7 provided an illustrative example of the approach introduced by Chapter 6. We developed a tool called PatternLens, which takes pre-processed production logs as input and generates a list of patterns detected in the logs, along with suggested changes. Providers can then accept or reject each pattern, contributing to the refinement of our approach.

This thesis is the first step toward the automation of web API evolution. We demonstrated the applicability of several techniques on web API usage logs, that effectively and efficiently identify the need for changes based on consumers' behavior.

2 Future Directions

The approach presented in this thesis may inform future studies and can be extended in several directions. It is a novel approach that suggests the analysis of web API usage logs for web API evolution. Thus, it can be considered a starting point that opens up new research opportunities for further investigation.

One direction for extension is analyzing changes in web API endpoint behavior, in addition to interface level changes. For instance, changes in response format or content, such as modifying the order of results, removing blank rows or columns from the response, or altering the default list of attributes returned. These changes might not break consumers' applications, but they can modify behavior and affect application performance without consumers' knowledge.

Secondly, the usability analysis performed in Chapter 5 can be expanded to include other usability attributes and sub-attributes. These attributes have already been characterized in this work as indicators that can be mainly tracked in usage logs. However, we only quantified indicators for the know-ability of web APIs.

Furthermore, the work presented in Chapter 6 can be continued by analyzing the usage logs generated after implementing the suggested changes. This will enable us to see if consumers are using the web API in the way we anticipated. We can also create migration scripts for the changes we suggest to assist consumers in upgrading. Additionally, an additional evaluation could be conducted to assess providers' benefits from applying our approach to their web API evolution.

Finally, the preliminary tool presented in Chapter 7 has the potential for further extension with additional functionality. Currently, the tool is designed to detect two specific types of patterns. However, future work could involve incorporating all of the patterns that we have defined.

Bibliography

[1] R. Abbas, F. A. Albalooshi, and M. Hammad. Software change proneness prediction using machine learning. In *2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT)*, pages 1–7. IEEE, 2020.

[2] S. L. Abebe, N. Ali, and A. E. Hassan. An empirical study of software release notes. *Empirical Software Engineering*, 21:1107–1142, 2016.

[3] A. Abelló Gamazo, C. P. Ayala Martínez, C. Farré Tost, C. Gómez Seoane, M. Oriol Hilari, and Ó. Romero Moral. A data-driven approach to improve the process of data-intensive api creation and evolution. In *CAiSE-Forum-DC 2017: CAiSE 2017 Forum and Doctoral Consortium Papers: proceedings of the Forum and Doctoral Consortium Papers Presented at the 29th International Conference on Advanced Information Systems Engineering (CAiSE 2017): Essen, Germany, June 12-16, 2017*, pages 1–8. CEUR-WS. org, 2017.

[4] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, 1993.

[5] D. Alonso-Ríos, A. Vázquez-García, E. Mosqueira-Rey, and V. Moret-Bonillo. Usability: a critical analysis and a taxonomy. *International journal of human-computer interaction*, 26(1):53–74, 2009.

[6] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. Van Deursen. Code smells for model-view-controller architectures. *Empirical Software Engineering*, 23:2121–2157, 2018.

[7] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pages 304–318, 2008.

[8] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. Introducing a ripple effect measure: a theoretical and empirical validation. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.

[9] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. A method for assessing class change proneness. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 186–195, 2017.

[10] R. K. Bandi, V. K. Vaishnavi, and D. E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE transactions on Software Engineering*, 29(1):77–87, 2003.

[11] B. Berendt, B. Mobasher, M. Spiliopoulou, and J. Wiltshire. Measuring the accuracy of sessionizers for web usage analysis. In *Workshop on Web Mining at the First SIAM International Conference on Data Mining*, pages 7–14. Citeseer, 2001.

[12] D. Berlind. What are apis and how do they work?, 2019.

[13] D. Bertram, A. Voida, S. Greenberg, and R. Walker. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 291–300, 2010.

[14] R. J. C. Bose, R. S. Mans, and W. M. Van Der Aalst. Wanna improve process mining results? In *2013 IEEE symposium on computational intelligence and data mining (CIDM)*, pages 127–134. IEEE, 2013.

[15] A. Brito, L. Xavier, A. Hora, and M. T. Valente. Why and how java developers break apis. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265. IEEE, 2018.

[16] T. Bush. How workarounds reveal the true needs of your api consumers, 2019.

[17] E. Carter. New research predicts continued growth in api testing market, 2018.

[18] D. Chapela-Campa, M. Mucientes, and M. Lama. Mining frequent patterns in process models. *Information Sciences*, 472:235–257, 2019.

[19] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[20] J. De San Pedro, J. Carmona, and J. Cortadella. Log-based simplification of process models. In *Business Process Management: 13th International Conference, BPM 2015, Innsbruck, Austria, August 31–September 3, 2015, Proceedings 13*, pages 457–474. Springer, 2015.

[21] C. R. de Souza and D. L. Bentolila. Automatic evaluation of api usability using complexity metrics and visualizations. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 299–302. IEEE, 2009.

[22] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

[23] B. Doerrfeld. 10+ api monitoring tools, 2018.

[24] H. Ed-Douibi, J. L. Cánovas Izquierdo, and J. Cabot. Example-driven web api specification discovery. In *Modelling Foundations and Applications: 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings 13*, pages 267–284. Springer, 2017.

[25] A. M. Eilertsen and A. H. Bagge. Exploring api: Client co-evolution. In *Proceedings of the 2nd International Workshop on API Usage and Evolution*, pages 10–13, 2018.

[26] T. Espinha, A. Zaidman, and H.-G. Gross. Web api growing pains: Stories from client developers and their code. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 84–93. IEEE, 2014.

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

[28] R. T. Fielding. *Architectural styles and the design of network-based software architectures.* University of California, Irvine, 2000.

[29] J. Gerken, H.-C. Jetter, M. Zöllner, M. Mader, and H. Reiterer. The concept maps method as a tool to evaluate the usability of apis. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 3373–3382. ACM, 2011.

[30] D. Gilling. 13 api metrics that every platform team should be tracking, 2022.

[31] N. Goel and C. Jha. Analyzing users behavior from web access logs using automated log analyzer tool. *International Journal of Computer Applications*, 62(2), 2013.

[32] W. Granli, J. Burchell, I. Hammouda, and E. Knauss. The driving forces of api evolution. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, pages 28–37, 2015.

[33] T. Grill, O. Polacek, and M. Tscheligi. Methods towards api usability: A structural analysis of usability problem categories. In *Human-Centered Software Engineering: 4th International Conference, HCSE 2012, Toulouse, France, October 29-31, 2012. Proceedings 4*, pages 164–180. Springer, 2012.

[34] C. W. Günther and W. M. Van Der Aalst. Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In *Business Process Management: 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007. Proceedings 5*, pages 328–343. Springer, 2007.

[35] L. P. Hattori and M. Lanza. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, pages 63–71. IEEE, 2008.

[36] A. Ivanchikj, I. Gjorgjiev, and C. Pautasso. Restalk miner: Mining restful conversations, pattern discovery and matching. In *Service-Oriented Computing–ICSOC 2018 Workshops: ADMS, ASOCA, ISYyCC, CloTS, DDBS, and NLS4IoT, Hangzhou, China, November 12–15, 2018, Revised Selected Papers 16*, pages 470–475. Springer, 2019.

[37] A. Ivanchikj, C. Pautasso, and S. Schreier. Visual modeling of restful conversations with restalk. *Software & Systems Modeling*, 17:1031–1051, 2018.

[38] R. Jagadeesh Chandra Bose and W. M. Van der Aalst. Abstractions in process mining: A taxonomy of patterns. In *Business Process Management: 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings 7*, pages 159–175. Springer, 2009.

[39] A. Joshi, K. Joshi, and R. Krishnapuram. On mining web access logs. *UMBC Computer Science and Electrical Engineering Department*, 1999.

[40] J. Kapusta, M. Munk, P. Svec, and A. Pilkova. Determining the time window threshold to identify user sessions of stakeholders of a commercial bank portal. *Procedia Computer Science*, 29:1779–1790, 2014.

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

[42] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló. 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. IEEE, 2020.

[43] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló. Improving web api usage logging. In *Research Challenges in Information Science: 15th International Conference, RCIS 2021, Limassol, Cyprus, May 11–14, 2021, Proceedings*, pages 623–629. Springer, 2021.

[44] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló. Patternlens: Inferring evolutive patterns from web api usage logs. In *Intelligent Information Systems: CAiSE Forum 2021, Melbourne, VIC, Australia, June 28–July 2, 2021, Proceedings*, pages 146–153. Springer, 2021.

[45] R. Kosala and H. Blockeel. Web mining research: A survey. *ACM Sigkdd Explorations Newsletter*, 2(1):1–15, 2000.

[46] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang. A systematic review of api evolution literature. *ACM Computing Surveys (CSUR)*, 54(8):1–36, 2021.

[47] M. Lamothe and W. Shang. When apis are intentionally bypassed: An exploratory study of api workarounds. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 912–924, 2020.

[48] M. Leemans and W. M. van der Aalst. Discovery of frequent episodes in event logs. In *Data-Driven Process Discovery and Analysis: 4th International Symposium, SIMPDA 2014, Milan, Italy, November 19-21, 2014, Revised Selected Papers 4*, pages 1–31. Springer, 2015.

[49] M. M. Lehman. Laws of software evolution revisited. In *Software Process Technology: 5th European Workshop, EWSPT'96 Nancy, France, October 9–11, 1996 Proceedings 5*, pages 108–124. Springer, 1996.

[50] M. M. Lehman and J. F. Ramil. Software evolution—background, theory, practice. *Information Processing Letters*, 88(1-2):33–44, 2003.

[51] V. I. Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[52] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister. Upset: visualization of intersecting sets. *IEEE transactions on visualization and computer graphics*, 20(12):1983–1992, 2014.

[53] J. Li, Y. Xiong, X. Liu, and L. Zhang. How does web service api evolution affect clients? In *2013 IEEE 20th International Conference on Web Services*, pages 300–307. IEEE, 2013.

[54] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, and M. Stocker. Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, pages 1–24, 2019.

[55] W. Maalej, M. Nayebi, T. Johann, and G. Ruhe. Toward data-driven requirements engineering. *IEEE software*, 33(1):48–54, 2015.

[56] A. Macvean, L. Church, J. Daughtry, and C. Citro. Api usability at scale. In *PPIG*, page 26, 2016.

[57] R. Malhotra and M. Khanna. Investigation of relationship between object-oriented metrics and change proneness. *International Journal of Machine Learning and Cybernetics*, 4:273–286, 2013.

[58] R. Malhotra and M. Khanna. An empirical study for software change prediction using imbalanced data. *Empirical Software Engineering*, 22:2806–2851, 2017.

[59] R. Malhotra and M. Khanna. Software change prediction: A systematic review and future guidelines. *e-Informatica Software Engineering Journal*, 13(1), 2019.

[60] M. Mathijssen, M. Overeem, and S. Jansen. Identification of practices and capabilities in api management: a systematic literature review. *arXiv preprint arXiv:2006.10481*, 2020.

[61] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi. Building more usable apis. *IEEE software*, 15(3):78–86, 1998.

[62] M. Medjaoui, E. Wilde, R. Mitra, and M. Amundsen. *Continuous API management.* " O'Reilly Media, Inc.", 2021.

[63] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[64] E. Mosqueira-Rey, D. Alonso-Ríos, V. Moret-Bonillo, I. Fernández-Varela, and D. Álvarez-Estévez. A systematic approach to api usability: Taxonomy-derived criteria and a case study. *Information and Software Technology*, 97:46–63, 2018.

[65] E. Murphy-Hill, C. Sadowski, A. Head, J. Daughtry, A. Macvean, C. Jaspan, and C. Winter. Discovering api usability problems at scale. In *Proceedings of the 2nd International Workshop on API Usage and Evolution*, pages 14–17. ACM, 2018.

[66] B. A. Myers and J. Stylos. Improving api usability. *Communications of the ACM*, 59(6):62–69, 2016.

[67] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 858–868. IEEE, 2015.

[68] S. Panichella, G. Canfora, and A. Di Sorbo. "won't we fix this issue?" qualitative characterization and automated identification of wontfix issues on github. *Information and Software Technology*, 139:106665, 2021.

[69] C. Pautasso, A. Ivanchikj, and S. Schreier. A pattern language for restful conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, pages 1–22, 2016.

[70] C. Pautasso and E. Wilde. Why is the web loosely coupled? a multi-faceted metric for service design. In *Proceedings of the 18th international conference on World wide web*, pages 911–920, 2009.

[71] M. Piccioni, C. A. Furia, and B. Meyer. An empirical study of api usability. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 5–14. IEEE, 2013.

[72] N. Pržulj, D. G. Corneil, and I. Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[73] G. M. Rama and A. Kak. Some structural measures of api usability. *Software: Practice and Experience*, 45(1):75–110, 2015.

[74] RapidAPI. State of apis report 2022, 2023.

[75] I. Rauf, E. Troubitsyna, and I. Porres. A systematic mapping study of api usability evaluation methods. *Computer Science Review*, 33:49–68, 2019.

[76] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[77] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *2011 27th IEEE international conference on software maintenance (ICSM)*, pages 303–312. IEEE, 2011.

[78] M. S. Santos, J. P. Soares, P. H. Abreu, H. Araujo, and J. Santos. Cross-validation for imbalanced datasets: Avoiding overoptimistic and overfitting approaches [research frontier]. *ieee ComputatioNal iNtelligeNCe magaziNe*, 13(4):59–76, 2018.

[79] A. Sarajlić, N. Malod-Dognin, Ö. N. Yaveroğlu, and N. Pržulj. Graphlet-based characterization of directed networks. *Scientific reports*, 6(1):1–14, 2016.

[80] T. Scheller and E. Kühn. Automated measurement of api usability: The api concepts framework. *Information and Software Technology*, 61:145–162, 2015.

[81] S. Serbout and C. Pautasso. An empirical study of web api versioning practices. In *International Conference on Web Engineering*, pages 303–318. Springer, 2023.

[82] S. Sohan, C. Anslow, and F. Maurer. A case study of web api evolution. In *2015 IEEE World Congress on Services*, pages 245–252. IEEE, 2015.

[83] M. Spiliopoulou, B. Mobasher, B. Berendt, and M. Nakagawa. A framework for the evaluation of session reconstruction heuristics in web-usage analysis. *Informs journal on computing*, 15(2):171–190, 2003.

[84] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web usage mining: Discovery and applications of usage patterns from web data. *Acm Sigkdd Explorations Newsletter*, 1(2):12–23, 2000.

[85] M. Srivastava, A. K. Srivastava, and R. Garg. Data preprocessing techniques in web usage mining: A literature review. In *Proceedings of International Conference on Sustainable Computing in Science, Technology and Management (SUSCOM), Amity University Rajasthan, Jaipur-India*, 2019.

[86] S. Suriadi, R. Andrews, A. H. ter Hofstede, and M. T. Wynn. Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. *Information systems*, 64:132–150, 2017.

[87] P. Suter and E. Wittern. Inferring web api descriptions from usage data. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 7–12. IEEE, 2015.

[88] W. Tan, Y. Fan, A. Ghoneim, M. A. Hossain, and S. Dustdar. From the service-oriented architecture to the web api economy. *IEEE Internet Computing*, 20(4):64–68, 2016.

[89] D. Tanasa and B. Trousse. Advanced data preprocessing for intersites web usage mining. *IEEE Intelligent Systems*, 19(2):59–65, 2004.

[90] G. Uddin and M. P. Robillard. How api documentation fails. *Ieee software*, 32(4):68–75, 2015.

[91] W. Van Der Aalst. Service mining: Using process mining to discover, check, and improve service behavior. *IEEE transactions on services Computing*, 6(4):525–535, 2012.

[92] W. Van Der Aalst. *Process mining: data science in action*, volume 2. Springer, 2016.

[93] W. M. Van Der Aalst. Challenges in service mining: record, check, discover. In *Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings 13*, pages 1–4. Springer, 2013.

[94] P. Vassiliadis, A. V. Zarras, and I. Skoulis. How is life for a table in an evolving relational schema? birth, death and everything in between. In *Conceptual Modeling: 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings 34*, pages 453–466. Springer, 2015.

[95] K. Vasudevan. What is api documentation, and why it matters?, 2017.

[96] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 319–328. IEEE, 2013.

[97] S. Wang, I. Keivanloo, and Y. Zou. How do developers react to restful api evolution? In *Service-Oriented Computing: 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings 12*, pages 245–259. Springer, 2014.

[98] D. Winer, S. Thatte, D. Box, G. Kakivaya, and A. Layman. SOAP: Simple Object Access Protocol. Internet-Draft draft-box-http-soap-01, Internet Engineering Task Force, Dec. 1999. Work in Progress.

[99] E. Wittern, A. T. Ying, Y. Zheng, J. A. Laredo, J. Dolby, C. C. Young, and A. A. Slominski. Opportunities in software engineering research for web api consumption. In *2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*, pages 7–10. IEEE, 2017.

[100] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering*, 21:2366–2412, 2016.

[101] L. Xavier, A. Hora, and M. T. Valente. Why do we break apis? first answers from developers. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 392–396. IEEE, 2017.

[102] T. Zhang, B. Hartmann, M. Kim, and E. L. Glassman. Enabling data-driven api design with community usage data: A need-finding study. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[103] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*, pages 318–343. Springer, 2009.

[104] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.

[105] M. F. Zibran, F. Z. Eishita, and C. K. Roy. Useful, but usable? factors affecting the usability of apis. In *2011 18th Working Conference on Reverse Engineering*, pages 151–155. IEEE, 2011.

Appendices

Appendix A

Appendix 1

1 Structural relationships derived from graphlets

Table A.1: Graphlets, the sequences they can generate, the spotted relationships between two nodes, and the inferred patterns.

Graphlets	Sequences	Relationships between nodes	Patterns
G1	A	Not connected node	-
G2	A,...,A	A node called consecutively k times	P1
G3	AB	Nodes called consecutively after one another	P2
G4	AB, BA, ABA	Nodes called in opposite direction	P3
G5	ABC	Nodes called consecutively after one another	P2
G6	BA, BC	Nodes that follow the same source node(s)	P4
G7	AB, CB	Nodes that are followed by the same target node(s)	P5
G8	ABCA, ABC, CA	Nodes called consecutively after one another Nodes called in opposite direction	P2, P3
G9	AC, ABC	A node being optional to another	P6
G10	ABCD	Nodes called consecutively after one another	P2
G11	ABC, DC	Nodes that are followed by the same target node(s)	P5
G12	AB, CD, CB	Nodes that follow the same source node(s) Nodes that are followed by the same target node(s)	P4, P5
G13	CD, CBA	Nodes that follow the same source node(s)	P4
G14	AD, BD, CD	Nodes that are followed by the same target node(s)	P5
G15	DB, DC, DA	Nodes that follow the same source node(s)	P4

G16	BDA, BDC	Nodes that follow the same source node(s)	P4
G17	ADB, CDB	Nodes that are followed by the same target node(s)	P5
G18	ABCDA, ABCD, DA	Nodes called consecutively after one another Nodes called in opposite direction	P2, P3
G19	ABCD, AD	A node being optional to another	P6
G20	AB, AD, CB, CD	Nodes that follow the same source node(s)	P4
G21	ABC, ADC	Nodes as possible P7s after another node	P7
G22	CBA, CDBA	A node being optional to another	P6
G23	CB, CDB, AB	A node being optional to another Nodes called consecutively after one another	P6, P2
G24	BA, BCD, BD	A node being optional to another Nodes called consecutively after one another	P6, P2
G25	CBA, CD, CBD	A node being optional to another Nodes called consecutively after one another	P6, P2
G26	ABCD, ABD	A node being optional to another	P6
G27	ABD, CD, CBD	A node being optional to another Nodes called consecutively after one another	P6, P2
G28	BCDBA, BCD, DB	Nodes called consecutively after one another Nodes called in opposite direction	P2, P3
G29	ABCDB, BCD, DB	Nodes called consecutively after one another Nodes called in opposite direction	P2, P3
G30	ABCDA, AC	Nodes called consecutively after one another A node being optional to another	P2, P6
G31	ABC, ACB, ADC	A node being optional to another Nodes as possible P7s after another node	P6, P7
G32	ABC, ADC, CA	Nodes as possible P7s after another node	P7
G33	BC, BAC, DC, DAC	A node being optional to another Nodes that follow the same source node(s)	P6, P4
G34	AB, AD, ACB, ACD	A node being optional to another	P6
G35	ABCD, AD, ACD	A node being optional to another	P6
G36	ABCD, ABCA, AD	Nodes called consecutively after one another A node being optional to another Nodes called in opposite direction	P2, P6, P3
G37	DABC, DC, ABCA	A node being optional to another	P6
G38	DC, DAC, DABC	A node being optional to another	P6
G39	BAD, BCD, BCAD	A node being optional to another Nodes as possible P7s after another node	P7, P6

G40	DBCAB, DCABC, DABCA	A node being optional to another Nodes that follow the same source node(s)	P6, P4
G41	ABCAD, ABD, ABCD	Nodes called in opposite direction	P3
G42	BC, BDC	A node being optional to another	P6
G43	DB, DAB, DACB, DCB	A node being optional to another Nodes that follow the same source node(s)	P6, P4

2 Interviews

Interview with DHIS2 WAPIs' consumers

Q1: Based on your experience using DHIS2 WAPI, share your opinion on its evolution (frequency, changes relevance, or practices followed by providers).
Q2: Do you communicate with WAPI providers? If yes, how?
Q3: Do providers take into account your requests when changing the WAPI?
Q4: Would you prefer having different ways of doing a task (or get some data), or just one way?
Q5: Have you ever used the following endpoints together when implementing a specific feature in your applications? (direct-follow, two-node loop, and feed-forward)
Q6: If it was available a unified endpoint that combines the two mentioned endpoints, would you use it, or would you continue using the individual ones? Why?
Q7: Can you give any examples on how you use the following endpoints? (examples from fork, inverted fork, and choice)
Q8: How would you react to the implementation of a new endpoint that would combine the two mentioned endpoints? Would you be open to use it?
Q9: Have you ever felt the need to make several calls of this endpoint in series, with different values for its attributes or parameters?
Q10: Do you think there may be a more efficient way to get/post the needed data only by making one call, instead of several ones?

Interview with FIB WAPIs' providers

Q1: How do you deal with the evolution of FIB WAPI? What is the starting point of a change?
Q2: How do you communicate with your consumers?
Q3: Do you take into account the feedback from consumers when planning the changes?

Q4: Are the following endpoints related in any way (e.g. usage, function, purpose)? (example from direct follow)
Q5: If consumers would call the previous endpoints together most of the time (one after the other always in the same order), would you consider this usage as an indication that there is a causal dependency relation between the two endpoints? Would this kind of relation encourage you to perform some action in order to improve these endpoints so that consumers can get the data more efficiently? Why yes/no?
Q6: Are the following endpoints related in any way? (example from the two-node loop)
Q7: If consumers would call the previous endpoints together most of the time, without following a specific order, would you consider this usage as an indication that these endpoints have combinable functions? Would this kind of relation encourage you to perform some action in order to improve these endpoints so that consumers can get the data more efficiently and simpler? Why yes/no?
Q8: Are the following endpoints related in any way? (examples from fork, inverted fork, and choice)
Q9: If most of the time consumers would call the previous endpoints after and/or the same set of endpoints, would you consider this usage as an indication that these endpoints might have a similar purpose or joinable functionalities? Would this kind of relation encourage you to reorganize these endpoints to simplify the data retrieval for consumers, by providing endpoints without duplicate functionality? Why yes/no?
Q10: Are the following endpoints related in any way? (example from the feed-forward)
Q11: If most of the time consumers would call the second endpoint optionally after the first endpoint, would you consider this usage as an indication that the second endpoint functionality might be supplementary or optional to the first one? Would this kind of relation encourage you to improve any of these endpoints so that consumers can get the data needed data more efficiently? Why yes/no?
Q12: If you see that consumers would call the following endpoint several times in sequence with different parameter values, would you consider this usage as an indication that there is a need to improve these endpoints so that consumers can get the same data with only one call? (example from the reflexive loop)