BRIDGING THE DIVIDE BETWEEN API USERS AND API DEVELOPERS BY MINING PUBLIC CODE REPOSITORIES

Maxime Lamothe

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy (Computer Science)
Concordia University
Montréal, Québec, Canada

September 2020

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: Maxime Lamothe

Entitled: **Bridging the Divide Between API Users and API Developers by Mining Public Code Repositories**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Liangzhu Wang

_____ External Examiner
Dr. Daniel German

_____ External to Program
Dr. Yan Liu

_____ Examiner
Dr. Yann-Gaël Guéhéneuc

_____ Examiner
Dr. Juergen Rilling

_____ Thesis Supervisor
Dr. Weiyi Shang

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

December 16, 2020 _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Bridging the Divide Between API Users and API Developers by Mining Public Code Repositories

Maxime Lamothe, Ph.D.
Concordia University, 2020

Software application programming interfaces (APIs) are a ubiquitous part of Software Engineering. Their evolution requires constant effort from their developers and users alike. API developers must constantly balance keeping their products modern and attractive whilst preserving their value and backward compatibility. Meanwhile, API users must continually be on the lookout to adapt to changes that could break their applications. In this thesis, we study existing literature to identify the state-of-the-art in API evolution research. We then use our findings to establish practical and scalable API evolution guidelines and tools to bridge knowledge gaps between API users and API developers. We base these guidelines and tools on public code repositories and use them to perform four empirical studies to further our understanding of API evolution.

To motivate this thesis, we first conduct a systematic literature review of the state-of-the-art in API evolution research. We find that there are a variety of unsolved challenges within the field, and that in particular, public code repositories have yet to be fully leveraged to aid both API users and API developers.

We then present three empirical studies that focus on helping API users deal with API knowledge gaps. We find that: (1) even if Android API documentation can help Android API users understand *what* to migrate from one API version to another, it does not often address *how* to do so; (2) although Android API migration knowledge is not always present in documentation, automated techniques can be created to mine this knowledge from existing public code repositories and automatically leverage it to help other API users; and (3) API misuse detection approaches can benefit from automatic example generation to reduce their false positive detection rates and improve their usefulness for API users.

Finally, we present an empirical study that uncovers general reasons why API users inquire about API workarounds and the solutions that are generally given to

them. Using this information, we develop three API workaround implementation patterns that can be used to detect instances of API workarounds in user code. We find that the knowledge contained within these detected instances of API workarounds can be fed back to API developers as opportunities to improve their APIs. Other API evolution research directions do exist; some are briefly covered in this thesis, for example identifying and using common API usage patterns, leveraging API migration patterns to inform API development, and API migration between programming languages; others still may yet be undiscovered. However, this thesis shows that it is possible to leverage existing knowledge contained in open-source repositories to help API users and API developers alike. This thesis opens the door to future research in the field of API evolution and demonstrates that it is possible to bridge knowledge gaps between API developers and API users for API migration, API misuse detection, and API workarounds usage.

Acknowledgments

I would first like to thank my supervisor Dr. Weiyi Shang for all of his advice and support throughout my Ph.D. I can unequivocally say that without him, this would not have been possible. Thank you for giving me this opportunity, and for all of the guidance you have given me throughout, I could not have asked for a better mentor.

I would like to thank all of my lab mates and collaborators throughout this journey, Muhammad Moiz Arif, Guilherme Bicalho de Pádua, Jinfu Chen, Hetong Dai, Zishuo Ding, Mehran Hassani, Mostafa Jangali, Zhenhao Li, Lizhi Liao, Nian Liu, Sophia Quach, Yuanjie Xia, Kundi Yao, Yi Zeng, Haonan Zhang, Dr. Tse-Hsun Chen, and Dr. Heng Li, it was an absolute privilege to have worked alongside you. I would also like to take this opportunity to thank everyone in the Concordia SE Group, professors and students, for the great presentations and stimulating conversations during my Ph.D.

I'm grateful for the support of my Ph.D. committee Dr. Yann-Gaël Guéhéneuc, Dr. Yan Liu, and Dr. Juergen Rilling for helping to guide me during this adventure.

I would also like to extend thanks to the members of my examination committee which includes my Ph.D. committee as well as Dr. Daniel German, for taking the time to evaluate my work.

To my family, thank you for always supporting me in my decisions and for believing in me, I would not be where I am today without you. To my friends, thank you for being there for me when I was available and for understanding when I wasn't.

I would like to extend a special thanks to Dirk Dubois for encouraging me to embark upon this journey and supporting me throughout. Finally, thank you to the love of my life, my fiancée, Erin, for your devotion and endless support.

I dedicate this thesis to my friends and family.

Related Publications

The work presented in this manuscript was published or submitted, with the author of this thesis as primary author and researcher, as listed below:

- **Bridging the Divide Between API Users and API Developers by Mining Public Code Repositories** (Chapter 1)**.** Maxime Lamothe. In Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020).

- **Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android** (Chapter 3)**.** Maxime Lamothe, Weiyi Shang. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR 2018), pp. 503-514.

- **A3: Assisting Android API Migrations Using Code Examples** (Chapter 4)**.** Maxime Lamothe, Weiyi Shang, Tse-Hsun Peter Chen. IEEE Transactions on Software Engineering (TSE 2020).

- **Assisting Example-based API Misuse Detection via Complementary Artificial Examples** (Chapter 5)**.** Maxime Lamothe, Heng Li, Weiyi Shang. Submitted to the International Conference on Software Engineering Technical Track (TSE 2021) in August 2020. **Under Review**

- **When APIs are Intentionally Bypassed: An Exploratory Study of API Workarounds** (Chapter 6)**.** Maxime Lamothe, Weiyi Shang. In Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Open-source software and software as a service led to the proliferation of application programming interface (APIs) development and usage. From millions of Android packages (APKs), to millions of open-source packages hosted by package managers such as Maven, PyPI, and npm, APIs have become an integral part of software development. APIs are used by millions of user applications to rapidly obtain existing functionality. Like traditional software, software APIs evolve and suffer from this evolution with documentation mismatches, misuses, unknown usage constraints, and performance and security issues. Meanwhile, user applications must also evolve to adapt to API changes. The current trends of software as a service and open public APIs present increasing opportunities and needs for developers to rely on externally maintained software. However, as a consequence, software developers become dependent on frameworks and public application program interfaces (APIs) [186].

Because the API users are typically independent from the development of an API, API users are at the mercy of the evolution of the interface. The development benefits provided by APIs come at a price. By relying on APIs, users inevitably couple some of their functionality to APIs over which they have little control [30]. This coupling can be a challenge to users as they are forced to deal with ever-evolving APIs [66]. Furthermore, knowledge gaps have been shown to exist between API developers and API users [54]. Often, API developers communicate about their APIs through one sided documentation channels such as wikis, manuals, tutorials, or API code examples [54]. API users have limited access to API developers and few channels to communicate their feedback. Some APIs even require knowledge of internal politics

to reliably get patches accepted [30]. Without a direct feedback channel, situations arise where API developers must rely on repeated user complaints to become aware of existing problems [54]. All too often, when users have issues with an API, for example needing a new feature or experiencing a run-time problem, users may choose to intentionally modify or work around the API [30]. These workarounds allow API users to obtain their desired functionality quickly and without going through potentially arduous communication with API developers. However, these workarounds are not without consequence. On the one hand, these workarounds are created by API users as temporary solutions, these workarounds become technical debt, endangering code quality and increasing future maintenance cost [216]. On the other hand, these API workarounds are potentially missed opportunities for API developers to improve their APIs (e.g., fixing bugs in the API).

In this thesis we propose to bridge the knowledge divide between API users and API developers by leveraging public code repositories. Therefore, whilst we seek to further existing research to provide aid to API users in adapting to API changes, we also seek to keep the API developers in the communication loop. Numerous API usages exist on public code repositories, and although some are erroneous, others are prime examples of quality API development and API usage. We believe that API users and developers alike can benefit from the knowledge ingrained in public code repositories. To overcome the divides that exist between API users and API developers we propose tools, techniques, and user studies to help API users and API developers. We seek to help API users adapt to changes to APIs (e.g., API migrations). Meanwhile we also seek to help API developers keep up with the ever-increasing demands from their users by finding examples of potential improvements hidden in API user code (e.g., API workarounds).

Prior research has produced tools that attempt to use existing documentation or historical code-change information to aid API developers' efforts to provide necessary information to API users [4, 48, 186]. We believe that public code repositories still contain vast untapped sources of knowledge to assist API users and API developers alike.

In this thesis, we first present a literature review of API evolution research. The information gathered in the literature review is used throughout the thesis to guide our research within the realm of API evolution. For example, prior research has had

success using public code repositories to learn more about API evolution. We therefore leverage public open-source software repositories to study and propose solutions to API evolution challenges faced by API users. Finally, we further leverage public open-source software repositories to study and propose solutions to API evolution challenges faced by API developers.

This chapter consists of the following parts: Section 1.2 presents the research hypothesis of this thesis. Section 1.3 presents an overview of the thesis. Section 1.4 briefly summarizes the contributions of this thesis. Section 1.5 presents the overall organization of the thesis.

1.1 Context

This section briefly presents the concept of APIs and presents an introduction to API evolution.

1.1.1 Definition of an API

To the best of our knowledge, the term application programming interface appeared for the first time in 1968 within the context of providing a remotely accessed, interactive computer graphics system [49]. Application programming interfaces (API) are varied and can encompass different concepts. For example, when the concept of information hiding was first coined by Parnas [204] in 1972, it was based on interfaces between modules, which today would likely fall under the umbrella of API.

Prior work has defined APIs as "the interface to a reusable software entity used by multiple clients outside the developing organization, and that can be distributed separately from environment code" [229]. Although the term 'API' can be used as a general term for an interface between software components, there exists nomenclature to refer to certain types of APIs. For example, software libraries [21, 32, 45, 61, 62, 81, 86, 89, 101, 121, 122, 170, 180, 220, 269, 304, 313], software frameworks [50, 51, 62, 65, 66, 90, 110, 111, 166, 184, 246, 264, 293–295, 297, 302, 312, 323], and Web services either RESTful [44, 157, 215, 248, 249, 286] or SOAP [249] have all been interchangeably been referred to as APIs because they all allow pieces of software to communicate, albeit in different ways. However, API terminology can sometimes be nuanced. For example, object-oriented languages, such as Java and C#, have specific keyword concepts to define interfaces [171, 203]. These interfaces are not necessarily APIs, however, according to the definition of an API presented by prior research [229]. Only if these interfaces are used by multiple clients (i.e., more than one API user application) outside of the developing organization, may they be considered APIs. In this paper, we use this API definition but also consider interfaces that may be used by multiple clients within a developing organization as APIs. An API within the context of this thesis would therefore be the interface to a reusable software entity used by multiple clients, and that can be distributed separately from environment code.

1.1.2 API Evolution

Prior studies have shown that APIs evolve for various reasons such as increasing complexity [161], and continuous change [64, 139]. However, due to their nature as a connection point between software modules, API evolution is not without side-effects. Many studies have shown the effects of API changes not only on the API itself [64], but also on its clients [162]. APIs may therefore change differently from traditional software artifacts. For example, Sun Microsystem preferred introducing the new interface *java.awt.LayoutManager2* rather than change the *java.awt.LayoutManager* because changing the latter would have broken existing code [250].

The evolution of APIs induces a variety of problems and challenges for API users and API developers alike [136].

On the one hand, as predicted by Lehman, *continuing change* [139] means that API developers must determine ways to keep their APIs useful, cutting edge, and competitive with other pieces of software [135] and API users must adapt to these API changes and new API releases. API version migration (i.e., the process by which API users modify their code to satisfy changes made from one API version to another while preserving behavior) can induce code modifications for API users because members of the API such as API classes, API methods, or API fields have been modified or removed. These modifications can render existing API user functionality non-functional. Because API changes can affect their users, the migration from one API version to another can require that users change API class invocations and usages, modify API method calls, and change how they use API fields. Prior work has found that field and method changes account for more than 80% of API migration cases [48].

On the other hand, *conservation of familiarity* [139] or existing API usages constrain the evolution of an API to avoid breaking changes while improving the API (e.g., security or performance improvements). API deprecation is one mechanism by which API developers can temporarily avoid breaking changes and inform their users that a specific API call will eventually be illegal as its functionality is slated to be removed from the API. This can give time to API users to migrate away from the deprecated functionality. The evolution of APIs therefore involves a balancing act of constant improvement and maintaining existing functionality. Maintaining existing functionality requires in-depth knowledge of use cases and architectural foresight and flexibility, while keeping up with rapid release cycles requires modifications to

user applications as well as learning about new APIs and changes to existing APIs. Therefore, when gathering literature for this thesis and its accompanying systematic review, we not only include work that directly studies APIs and their evolution, we also consider work that focuses on finding solutions to problems caused by API evolution.

1.2 Research Hypothesis

By decoupling and bundling useful functionality, APIs provide unequivocal benefits to software engineering practices. However, through this very accomplishment, APIs also introduce a coupling between their users and developers. Although this coupling allows API users and developers to share an interface, it does not promote communication between them. Therefore, knowledge gaps can form between API users and API developers. These knowledge gaps can obfuscate how to properly use an API for users and prevent API developers from understanding what their users expect from their API.

Thesis Statement: *The knowledge gaps between API users and API developers are an important part of API evolution issues. Public code repositories can be leveraged to bridge the knowledge gaps between API users and API developers and mitigate these issues.*

The goal of this thesis is therefore to help with the usage, and development of APIs. We believe that we can leverage open-source software through data-mining practices to better understand the nature of the problems introduced by API evolution, and to uncover solutions to mitigate the problems faced by both API users and API developers. This thesis presents evidence that it is indeed possible to leverage public code repositories to aid API developers, particularly with Android API migration, API misuse detection in Java, and API workarounds. This thesis is divided into two parts, the first part aims to help with API usage through: 1) an empirical study of the challenges of API migration, 2) an approach to help API users with the API migration process, and 3) an approach to improve API misuse detection; the second part of this thesis aims to support API development by uncovering why and how API users use API workarounds, and how these workarounds can be leveraged to improve APIs.

1.3 Thesis Overview

This section presents a brief overview of this thesis.

1.3.1 Chapter 2: Literature Review

In Chapter 2, we present a systematic literature review of API evolution literature. We systematically uncover publication trends, from the most popular API evolution subtopics, to the most popular APIs used to evaluate API evolution research. We identify the state-of-the-art in various subtopics of API evolution research such as dealing with breaking changes, reducing API misuses, and improving API usability. Furthermore, we also identify current and future challenges within the field of API evolution. In particular we find that leveraging and mastering the feedback systems involved in API evolution [140] is the next hurdle to overcome to attain proper API engineering. We use the findings uncovered in this literature review to motivate the work in the rest of this thesis. In particular, we help API users and API developers by providing tools to aid them leverage feedback systems to aid with Android API migration, Java API misuse detection, and API workarounds.

1.3.2 Chapter 3: What are the Challenges Associated with API Migration?

In Chapter 3, we present an exploratory of API evolution on the Android API. We namely concentrate on documentation and historical code-changes and their involvement in Android API migration. These artifacts are prime examples of official communication channels between API users and API developers. The Android API documentation is the primary source of official information for API users that seek to understand the API and historical code-change information provided by API developers can also allow API users to determine how an API might be used, should documentation prove insufficient. As a first step, we opt to leverage the Android documentation because prior work has shown the importance of documentation in API evolution [233]. As a second step, we leverage the official Android historical code change information (e.g. commits) to determine their corroboration with the results of the Android API migration pathways obtained from documentation in the previous step. In a third step we use free and open source apps and manually migrate

them using the identified migration pathways; we leveraged the Android API migration suggestions that we automatically recovered from both documentation (including official online documentation, commit message, and code comments) and historical code change information.

We find that the information needed to identify replacement API methods for Android API migrations often resides explicitly in online documentation and the official Android API repository commits as natural language text. Our findings indicate that although not all migrated methods can be found in the official Android online documentation, information needed to assist in API migration can be also found in other forms of documentation, such as code commit messages and code comments. Therefore, in the case of the Android API, the challenge of API migration is not to determine what the API user must migrate, because that information is usually available within official communication channels.

Furthermore, although API documentation has been generally reported as incomplete or outdated [52], Android API users should still consider the official documentation of the Android API as their primary source of information. Our results show that current practices can allow API users to obtain adequate migration information from Android API developers to understand *what* to migrate. However, there are still problems in bridging an understanding on *how* to migrate APIs.

1.3.3 Chapter 4: Using Existing API User Knowledge as API Migration Aid

In Chapter 4 we propose an approach, named A3, that mines and leverages source code examples to assist API users with understanding "how" to apply Android API migrations. We focus on Android API migrations due to Android's wide adoption and fast evolution [162]. Our approach automatically learns the API migration patterns from code examples taken from available code repositories, thereby providing varied example patterns. Afterwards, our approach matches the learned API migration patterns to the source code of the Android apps to identify API migration candidates. If migration candidates are identified, we apply the learned migration patterns to the source code of Android apps, and provide the resulting migration to developers as a potential migration solution. Our approach is particularly beneficial to less knowledgeable API users who might benefit from API migration data that can be

obtained from domain experts. Our approach can automatically identify 83 migration patterns with 96.1% precision in Android APIs, and obtains a recall of 97% using a seeded repository. Based on 80 migrations candidates in 32 open source apps, our approach can generate 14 faultless migrations, 21 migrations with minor code changes, and 36 migrations with useful guidance to developers. Through a user study with 15 participants and 6 API migration examples, we show that our approach provides, on average, a 29% migration time improvement and is seen as useful by our participants. Our approach can be adopted by Android app developers to reduce their API migration efforts to cope with the fast evolution of Android APIs. Our approach also exposes the value of using the knowledge that resides in code examples to provide assistance to API users with API evolution.

1.3.4 Chapter 5: Improving Misuse Detection Approaches

In Chapter 5, to further aid API users, we conduct an exploratory study of API misuses identified by a state-of-the-art API misuse detection tool [4]. Current API misuse detection tools are meant to be used by API users to automatically detect instances where they might have misused an API in their code [5]. However, these tools currently present high false-positive detection rates [4], which hampers their usefulness to API users. We therefore particularly concentrate on falsely detected misuses to uncover reasons for their detection. Through our manual study of real examples of falsely detected API misuses, we uncover 108 cases where correct API usages were falsely identified as API misuses out of 384 manually verified cases. We classify these cases as alternate but correct API usages. From these alternate correct API usages, we discover five patterns that can be used to transform existing API usage examples into artificial API usage examples. These artificial examples can be used to cover the knowledge gaps caused by undiversified API usage examples, and thereby decrease the false-positive detection of API misuses. We find that using these artificial API usage examples does not reduce the recall of existing state-of-the-art misuse detectors, and allows for the removal of some falsely identified API misuses.

1.3.5 Chapter 6: Guiding API Development by Using API Workarounds

In Chapter 6, to enhance feedback channels from API users to API developers, we conduct an exploratory study of API workarounds requested and implemented by API users. We manually examined 400 posts from Stack Overflow, where we found that API users request API workarounds for a variety of reasons, such as dependency issues, missing functionality, and runtime problems. These reasons are valuable for API developers because gaining access to these workarounds could improve their APIs. Furthermore, we identified answers accepted by API users who request API workarounds. By studying these answers, we found that carrying out such API workarounds may not be a trivial task. In particular, a majority of API workaround solutions require special implementations to bypass the API. Therefore, because API users are willing to go through the trouble of using special cases to obtain their desired functionality, we can assume that the functionality must be of some value to them, and might therefore benefit from the evaluation of the API developers.

To follow up on our exploratory study, we study workaround implementations that are suggested in the Stack Overflow posts, and we observe three API workaround patterns. The knowledge contained in the implementation of these patterns can help API developers improve their API by adding desirable features, fixing unexpected behavior, and improving backwards compatibility.

Because our API workaround patterns were uncovered using forum questions and answers, we seek to confirm their existence in real-life API user code and confirm their usefulness with API developers using five open-source APIs. We confirmed the existence of the three patterns of API workarounds that we had previously identified, in open-source GitHub projects. Finally, we submitted and observed 12 feature requests to developers based on the API workarounds to improve the APIs. Among these requests, five are already closed, and six more have been confirmed by API developers as bugs or missing features. Our study and findings highlight the value of studying API workaround usages as a means to bridge the gap between API developer and API users to assist API evolution. In particular our results show that it is possible to use API usage data to help API developers understand some of the features that users would like, or bugs that could be fixed.

1.4 Thesis Contributions

This thesis demonstrates that public code repositories provide valuable untapped potential sources of knowledge for both API users and developers. We propose approaches to leverage public code repositories to extract both API user and API developer knowledge and find that this knowledge bridges divide between API users and developers and can help assuage problems such as API migration for Android API users, high rates of false positives in API misuse detection, and API workarounds.

In particular our contributions are:

1. We demonstrate that when migrating from one Android API version to another, understanding *what* to migrate does not necessarily help API users understand *how* to migrate.

2. We propose an approach that extracts API migration knowledge from existing Android app public code repositories to help less knowledgeable Android API users migrate their applications.

3. We show that it is possible to uncover and leverage general API usage patterns to improve API misuse detection for Java APIs.

4. We show that API developers can use API user public code repositories to understand why API users use workarounds and to leverage this information to improve APIs.

1.5 Thesis Organization

The rest of this thesis is organized into four main parts:

- **Part I:** Contains Chapter 2 which presents a systematic literature review of API evolution research.

- **Part II:** Contains research aimed at helping API users. Namely:

  - Chapter 3 presents an exploratory study of Android API migration and identifies challenges to API migration.

  - Chapter 4 presents an approach, motivated by Chapter 3, to extract API migration knowledge from existing public code repositories to help less knowledgeable API users migrate their applications.
  - Chapter 5 presents API usage example patterns to complement and improve existing state-of-the-art API misuse detection approaches.

- **Part III:** Contains research aimed at helping API users. Specifically:
  - Chapter 6 concentrates on API developers, and presents an empirical study into API workarounds and provides approaches to leverage API workarounds to improve APIs.

- **Part IV:** Contains Chapter 7 which concludes this thesis and highlights the major contributions of this thesis as well as avenues for future work

Part I

Literature Review

Chapter 2

Systematic Literature Review of API Evolution Research

API evolution has been extensively studied in prior research and many challenges to both the development and usage of APIs have been uncovered. While plenty of these challenges have been studied, many still remain. However, to the best of our knowledge, these challenges are scattered in the literature, which hides advances but also cloaks important, remaining challenges. In this chapter, we provide a systematic review of existing API evolution literature. Within this review we uncover publications trends in the field of API evolution, as well as trending subtopics within the field. We uncover common research goals within API evolution research as well as common evaluation methods, metrics, and subjects. We highlight the current state-of-the-art in API evolution research and provide an overview of known existing challenges for API evolution research as well as new challenges that were uncovered during this literature review. We conclude that the main remaining challenges related to API evolution are to automatically identify and leverage factors that drive API changes, create and use uniform benchmarks for API research evaluation, and to understand the sweeping impacts of API evolution with respect to API developers, API users, and with respect to various programming languages. Within the context of this thesis, we concentrate on the impacts of API evolution that can create knowledge gaps for API users and developers such as API migration, and API misuses. We also leverage factors that could drive API changes (e.g., API workarounds). We rely on the information uncovered in this systematic literature review to create tools and approaches that remedy

to some the challenges that we uncovered. Some of the challenges uncovered in this chapter remain open to future work.

2.1 Introduction

Software application programming interfaces (APIs) allow their users to save time and effort by relying on pre-made functionality [192]. It is therefore no surprise that APIs that provide desirable functionality are often used by software developers and that their usage is highly recommended to promote software quality while reducing development effort. For example, the Android API allows over 8 million APKs [88] in the Google Play store alone, to run on mobile devices across the world.

APIs are by definition interfaces and meant to be used as entry points to reusable software entities [229]. They are not independent, single software entities but are instead packaged with and offered by software libraries [62], frameworks [111], or web services [248].

The ease with which APIs can be discovered and used has grown in line with the advent of software as a service [135] and the growth of open-source software repositories like GitHub. For example, jUnit, a popular unit testing framework, has been used by over 20,000 applications in a 42,000 application sample [242], and is often adopted by users migrating away from other test frameworks (e.g., testng) [56]. APIs are endowed with an undeniable potential to impact software development. Understanding, mitigating, and leveraging the impacts of APIs on software development is therefore crucial to effectively design and use software APIs [232].

For the last few decades, interest in APIs has grown rapidly in the software research community. As interest grew, so did the number of publications related to APIs. We only identified one work published in 1994 with a relation to API evolution. Meanwhile, we identified 49 works related to API evolution in 2018. Researchers explored a variety of aspects of APIs, from API usability and misuses, to API maintenance, migration, documentation, recommendation and more. Software API research showed that APIs are not simple useful artifacts, nor can they easily be used and forgotten. APIs, like other software artifacts, evolve over time or suffer consequences [161]. In this thesis we provide a systematic review of the literature related to API evolution and the impacts and challenges imposed by that evolution. *To*

the best of our knowledge, these challenges are scattered in the literature, which hides advances but also cloaks important, remaining challenges.

Because APIs are inherently software artifacts, they are not immune to Lehman's laws [139]. To remain useful and competitive, APIs must change and therefore evolve. API evolution can cause various issues for both their users and their developers [135, 161, 192, 231]. We found many papers related to API evolution, which we study and present in the rest of this paper. Due to both the breadth and depth of research related to API evolution, it is difficult to determine the extent of prior research, for example, which problems were uncovered, and which solutions were proposed. Therefore a survey of prior work between 1994 and 2019 (i.e., 25 years) related to API evolution would benefit the software research community as well as software developers, by highlighting existing research into API evolution, presenting the current state-of-the-art solutions to challenges that were uncovered, and by enumerating challenges that have yet to be solved.

Prior research has produced many empirical studies [24, 111, 116, 162], new tools and techniques [168, 229, 232], and datasets [3, 19, 242] to uncover and solve the issues caused by API evolution. In this thesis we seek to define and decipher the state of the field of API research. We study prior research to uncover and summarize the motivation, methodology, evaluation, and results of prior API evolution research. We also summarize the current state-of-the-art in API evolution research to aid future research development and comparison. Furthermore, we also uncover unresolved challenges in prior research and present future research avenues in the field of API evolution.

Section 1.1 defines APIs and API evolution for this paper. Section 2.2 presents the methodology used to find the papers selected for this literature review. Section 2.3 highlights the various goals, tools, and evaluations that have evolved in API evolution research. Section 2.4 presents the state-of-the-art in API evolution research. Section 2.5 presents open API evolution challenges that remain either partially or completely unsolved by current research. Finally, Section 2.6 concludes the paper.

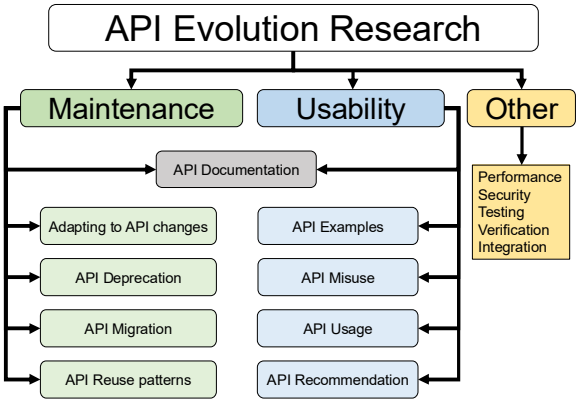


Figure 1: API research topics

2.2 Methodology

We used a well-defined, structured and systematic approach to produce a survey on API evolution. The approach followed was inspired by guidelines from Kitchenham et al. [129] and Petersen et al. [207].

2.2.1 Research questions

The goal of this systematic literature review is to provide a structured and categorized aggregate of existing API evolution research to uncover the state of API research. This knowledge will hopefully allow insight into the current state-of-the-art research and provide a quick reference into existing practices and currently unsolved challenges for future research. To achieve this goal, we designed the following research questions (RQs):

– *RQ1: How has the field of API evolution research evolved?*

  We explore papers related to API evolution, we provide an overview of these paper, categorize them, identify their goals, and investigate strategies used by API evolution researchers to evaluate their findings and discuss evaluation trends. We present our findings for this RQ in Section 2.3.

– *RQ2: What is the current state-of-the-art in API evolution research?*

  We present state-of-the-art approaches and tools proposed to deal with problems related to API evolution. We present our findings in Section 2.4.

Table 1: Publications found by search engine, after filtering

Search Engine	**Publications**	**Cross-Referenced**
ACM Digital Library	157	45
Elsevier Science Direct	12	10
IEEE Xplore Digital Library	27	26
Springer Online Library	29	27
Wiley Online Library	3	3
Google Scholar	847	111
Total *(duplicates removed)*	964	111

– *RQ3: What are the current and future challenges related to engineering APIs?*

Finally, we uncover current and future challenges left to solve for future API research. We select the tools and studies presented in this thesis solve challenges from this list. We present our findings in Section 2.5.

2.2.2 Literature repository selection

We used prior state-of-the-art software engineering literature reviews [113, 119] to obtain our selection criteria for online literature repositories. Our original selection of papers came from the following technical publishers:

- ACM Digital Library
- Elsevier Science Direct
- IEEE Xplore Digital Library
- Springer Online Library
- Wiley Online Library

We augmented our paper selection by performing a search in the Google Scholar database with "API evolution" as a search string. This was done to supplement the selection of papers from technical publishers and to ensure the widest possible search scope for our survey. Furthermore, we mined the references of the papers in our original selection, using forward and backward snowballing, to find cited works that contained abstracts that appeared to present work within the scope of API evolution.

2.2.3 Literature search and selection

Using our predefined literature repositories, we performed searches using the "API evolution" search phrase[1]. We further constrained the search to the fields of *computer science* and *software engineering* for the Springer, Wiley, and Elsevier online libraries. The results obtained are presented in Table 1. The results highlight the absolute number of publications found in each library, as well as the number of publications that were cross-referenced and available in multiple libraries. After accounting for all duplicate publications, we found a total of 964 publications (1075 before removing duplicates). We then filtered the results of this search, keeping only results which met the following criteria:

1. Document must be written in English
2. Document must be related to the field of computer science or software engineering
3. Document should have a relation to API evolution
4. Document must not be a Master or PhD thesis
5. Document must be fully available from one or more online library

A flowchart of our publication selection process can be found in Figure 2. Based on our filtering process, we obtained 108 publications. After checking the references of the chosen papers, we added a further 183 publications to the survey. These papers were likely missing in the initial library search due to nomenclature differences (i.e. *Framework* evolution instead of *API* evolution). Therefore, using the results of our initial search, along with any references that matched our filtering criteria, we obtained a total of 291 publications (or *primary studies*) with which to conduct this survey.

While we concede that it is unlikely that we managed to find and present all of the papers linked to the topic of API evolution in this study, we believe that the sample of publications chosen for this study is representative of the state of the art in the field of API evolution. We are confident that the majority of published works in the field of API evolution are present in this study and that the trends and findings in this work are the state-of-the-art.

[1]The latest search was conducted on September 2nd, 2019.

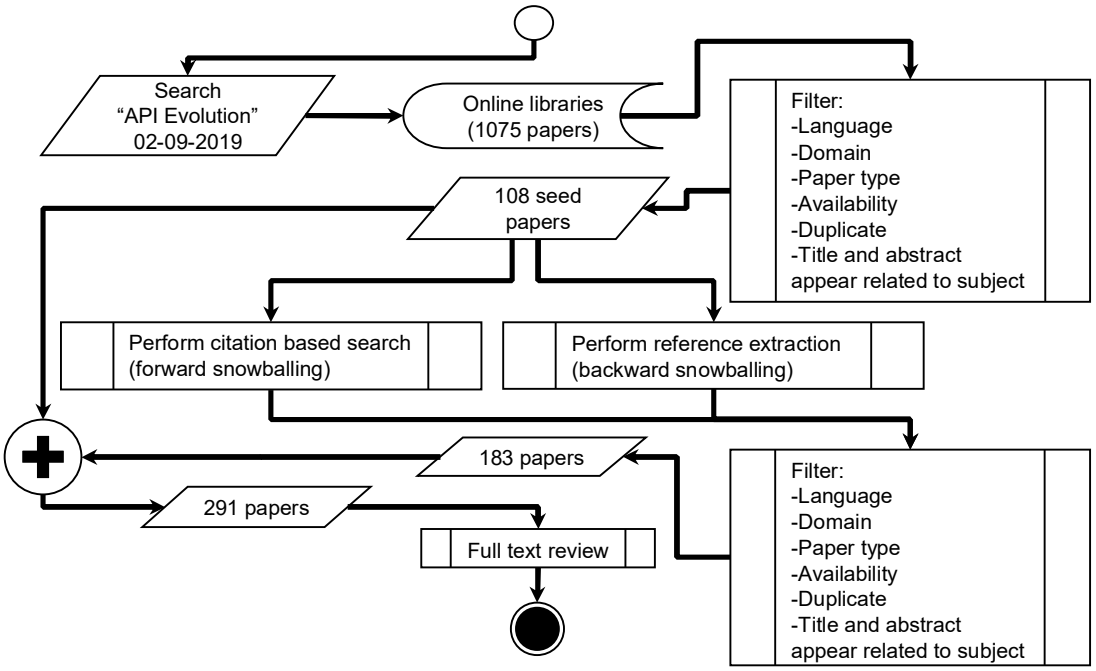


Figure 2: Publication selection process

2.2.4 Data extraction and collection

To answer our research questions, we carefully examined and extracted information from each of the 291 publications selected for this study. We paid particular attention to the motivation, contributions, methodologies and tooling, results, and challenges presented in the publications. To present concise and practical information, we categorized our findings into abstract categories whenever possible. The types of data extracted from each publication and their relevance to each research question are presented in Table 2. We did not include the year 2019 in the yearly trends presented in Section 2.2.5 because the year was not fully complete during the writing of this study, and therefore would not have presented a fair comparison to previous years. However, all papers that met our filtering criteria and were officially published as of September 2nd, 2019 were included in the other parts of this study.

2.2.5 Overview of primary studies

To answer RQ1, we categorize the topics of our selected publications, determine publication trends in the field of API evolution, and uncover publication patterns in API evolution research. We look at which researchers and organizations publish

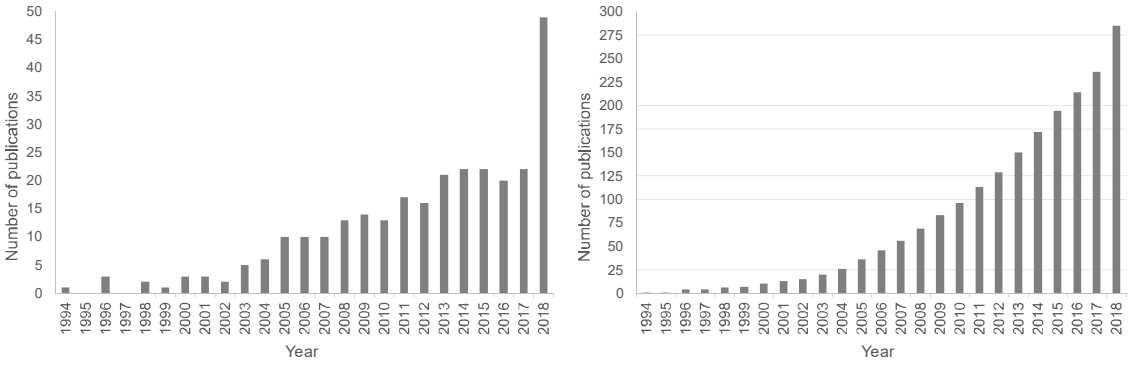
Table 2: Data extracted for our research questions

RQ	Type of Data Extracted
RQ1	Title, author information (names and affiliations), publication information (type, year, and location), names and sources of systems under test, types of evaluations performed, evaluation metrics, study motivation, methodology, and paper type
RQ2	Paper type, primary contribution, challenges uncovered and solved
RQ3	Unresolved questions, future research avenues

most in the field, how often papers are published, in which type of venue they are published, and with which type of work they are most related. We also categorize API evolution papers into five contribution types *Datasets*, *Empirical studies*, *New tools and techniques*, *Proposals and reports*, and *Surveys*.

Publication trends

Publications in API evolution are trending upwards. As shown in Figure 3, the number of publications with topics related to API evolution more than doubled from 2017 to 2018. Furthermore, we can also observe an exponential increase in the number of cumulative publications per year. API evolution is not only an active research topic, but is also a growing research field.



(a) Publications per year (b) Cumulative publications per year

Figure 3: API evolution publications from Sept 19th 1994 to Dec 31st 2018

Researchers and Organizations

Table 3 presents the top 8 researchers by publications related to API evolution (Top 5 positions including ties). The authors considered in Table 3 are either primary authors or co-authors of papers in our sample of 291 papers. Tien N. Nguyen is the most frequent contributor to API evolution research followed by Martin P. Robillard, and Anh Tuan Nguyen. The top co-authors of API evolution research are for the most part affiliated with universities in the U.S.A, with the notable exceptions of Martin P. Robillard who is affiliated with McGill University in Canada, Marco Tulio Valente from Federal University of Minas Gerais in Brazil, and Romain Robbes with the Free University of Bozen-Bolzano in Italy. Leading researchers in the field of API evolution are globally distributed, with a particular concentration in the U.S.A.

Geographical Distribution of Publications

To determine the geographical location of publications, we considered the country of the institution affiliated with the first author of each work. As shown in Table 4, we can see that the USA, with 35.4% of publications, is the country with the primary number of works in the field of API evolution, followed by Canada and China with 17.2% and 8.6% of publications respectively.

Most Common Publication Venues

The publications studied in this paper are spread over a variety of venues, some are more popular than others. Amongst the reviewed publications, the most common venue for journal paper publications is IEEE's Transactions on Software Engineering (TSE) with 8 journal publications, followed by Empirical Software Engineering with 6

Table 3: Top Co-authors on the subject of API evolution

Name	Affiliation	Country	Publications
Nguyen, Tien N.	University of Texas at Dallas	U.S.A	22
Robillard, Martin P.	McGill University	Canada	17
Nguyen, Hoan Anh	Iowa State University	U.S.A	12
Valente. Marco Tulio	Federal University of Minas Gerais	Brazil	12
Nguyen, Anh Tuan	Iowa State University	U.S.A	11
Robbes, Romain	Free University of Bozen-Bolzano	Italy	11
Dig, Danny	Oregon State University and University of Illinois	U.S.A	10
Kim, Miryung	University of California, Los Angeles	U.S.A	10

Table 4: Geographical distribution of papers

No.	Country or Region	Papers
1	USA	103
2	Canada	50
3	China	25
4	Germany	20
5	Brazil	13
6	Netherlands	10
7	Australia	8
8	Switzerland	7
9	Czech Republic	7
10	Belgium	6
11	Italy	5
12	France	4
13	South Korea	4
14	Finland	3
15	Greece	3
16	Spain	3
17	Sweden	3
18	Chile	2
19	Japan	2
20	New Zealand	2
21	Austria	1
22	Denmark	1
23	Ethiopia	1
24	Hong Kong	1
25	India	1
26	Norway	1
27	Portugal	1
28	Russia	1
29	Singapore	1
30	United Arab Emirates	1
31	United Kingdom	1

Table 5: API publication by type

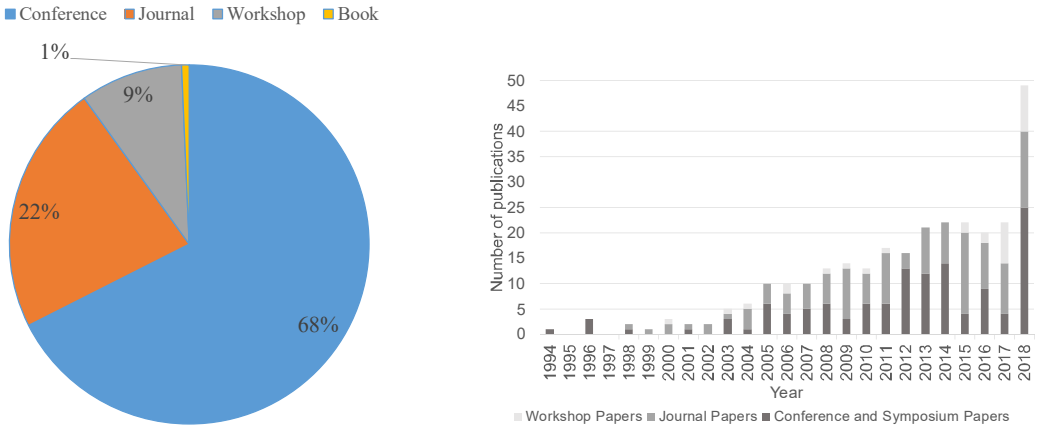
Publication Type	Papers
Conference	196
Journal	66
Workshop	27
Book	2

Table 6: API publication contributions

Main contribution	Papers
New Tools and Techniques	175
Empirical Studies	100
Proposals and Reports	8
Surveys	5
Datasets	3

API evolution journal publications. The most common conference is the International Conference on Software Engineering (ICSE) with 42 publications. The most common workshop is the Workshop on API Usage and Evolution (WAPI) with 12 publications (All of the publications published at WAPI).

As shown in Figure 4 we can see that the majority of publications in API evolution are conference papers, followed by journal papers and workshops, with only a slim minority (2) of books being published. We can also see that workshop papers appear to be increasing in numbers starting in 2017. This increase in workshop publications is likely due to the founding of the International Workshop on API Usage and Evolution (WAPI) in 2017. However, WAPI did not occur in 2019 or 2020, and trends might therefore experience a shift in those years as a result.



(a) Overall venue distribution

(b) Yearly venue distribution

Figure 4: API evolution publication venues

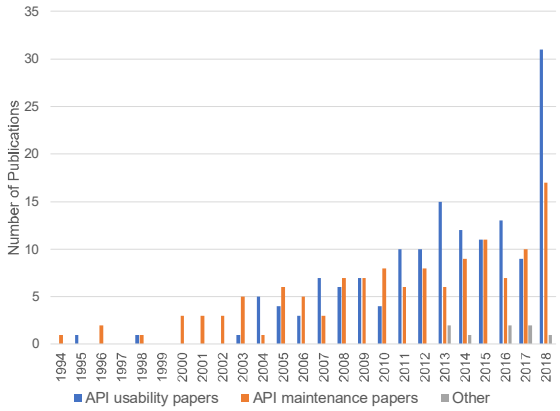
Publication Topics

We classified the 291 publications into various topics through the use of keywords provided by the authors within the papers themselves, keywords provided by the publisher (e.g., IEEE Keywords), or through the use of our judgment in cases where we could not recover relevant keywords.

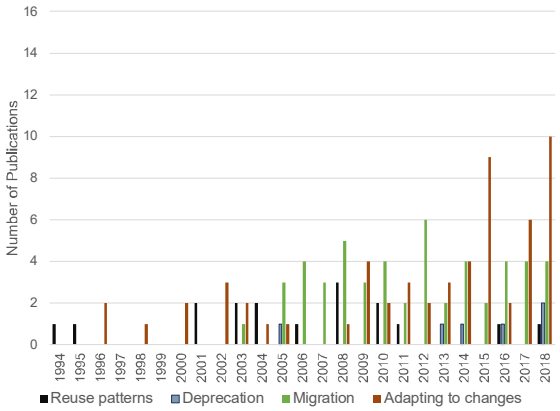
We first employed closed card sorting to sort papers into three blanket categories, *API Maintenance* which contains 130/291 publications, *API Usability* which contains 150/291 publications, and *Other* which contains 11 publications. We then used a second round of closed card sorting to further subdivide each blanket category as shown in Figure 1. Since the *Other* category only contains 11 publications of various topics, it was not subdivided into subcategories. The evolution trend of the three categories and their subcategories can be observed in Figure 5. The state-of-the-art, inception, and trends of the various topics of API research are further discussed in Section 2.4.

Publication Contribution Types

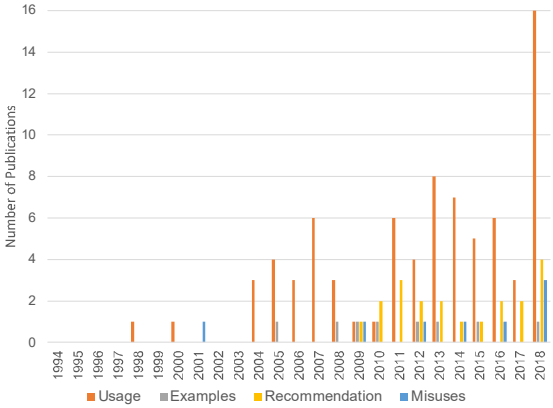
We also classified our sample of 291 publications into five publication contribution types using an open cart sorting approach. We rely on the judgement of the reviewers to extract the primary contribution of each paper. It is possible for a paper to present more than one contribution, and we sometimes must rely on human judgement to identify the primary or main contribution. Similarly to the research topic classification in Section 2.2.5, we created the contribution type categories by using author and



(a) API Maintenance and Usability publication trends



(b) API Maintenance subtopic publications per year



(c) API Usability subtopic publications per year

Figure 5: API Maintenance and Usability subtopic publication trends

publisher keywords, while also relying on publication venue information when it was relevant. These sources of information were combined with our best judgement to classify each publication after reading it. We obtained the following five contribution types: *New Tools and techniques*, *Empirical studies*, *Proposals and reports*, *Surveys*, and *Datasets*. The overall classification of the publications we studied can be found in Table 6.

2.3 The Evolution of API Evolution Research

2.3.1 API evolution research goals

We continue to answer RQ1 by presenting the various goals that we uncovered when surveying API evolution literature. API evolution presents various avenues for research. For example, it is possible to empirically observe the impact of API changes on API users [135], otherwise known as the effect of *perceived complexity* on users [139]. These studies can then provide motivation and insight to develop software tooling [52]. For example, within this thesis, we observe that Android API users have a hard time understanding how to migrate from one API to another, while they are not as challenged when attempting to determine what to migrate. In order to better understand the trends in API evolution research, we use our publication contribution classification of the 291 papers. We divide this section using the five contribution types identified in Section 2.2.5, namely, *Datasets*, *Empirical Studies*, *Proposals and Reports*, *Surveys*, and *New Tools and Techniques* to uncover which primary contributions align with which research goals.

New Tools and Techniques

As shown in Table 6, the majority of the papers present new tools and techniques to help with API evolution. These tools and techniques vary in scope and purpose, some attempt to reduce complexity [232], others attempt to help conserve familiarity [233], and others attempt to help organize changes [139, 245]. However, they all seek to resolve problems caused by API evolution through the intervention of either a tool or a new technique. We use existing surveys on API property inference techniques [229], recommendation systems [232], and software merging [168] as well as some of our own categorizations to label our dataset into API research topics presented in Section 2.2.5 of this survey (i.e., adapting to API changes, documentation, deprecation, examples, misuse, migration, recommendation, reuse patterns, usage, and other).

Tools and new techniques related to API evolution typically seek to present tools and techniques to help with API evolution by resolving problems that API evolution can cause for API users (e.g., API migration tools), or even to help reduce the development burden on API developer (e.g., automatic API documentation tools).

Empirical studies

The second largest category of API evolution publications are presented as empirical studies. The empirical studies we observed within our dataset can largely be divided into three sub-categories. Data-mining empirical studies, that make use of data from several projects or non-human sources, empirical case studies, which target specific projects and often provide in-depth results for a few specific non-human sources, and finally user studies which make use of human participants.

Data-Mining studies: Data-mining studies concentrate on using large sources of data to provide evidence for the existence of problems and to determine their impact. For example, mobile apps and the android API evolves quickly [162]. Therefore, many questions arose about the impact of the rapid evolution on clients [162], app categories [96], compatibility problems [98], and more [24, 39].

Case studies: Case studies study a few (e.g., fewer than 10) systems. Comparatively to data-mining studies, the results of case studies are specific to the studied systems. These studies present a range of goals, from determining the impact of API evolution on API users [111], determining whether IDEs influence the usability of dynamic and static APIs [208], determining the factors that support the long term success of frameworks [184], and many more [16, 288, 289].

User studies: We classified eight of the papers reviewed for this systematic literature review as user studies. These papers rely on human responses to answer their research questions, which have a strong usability component. Therefore, we surmise that user studies are particularly well suited for API usability studies. The papers determine learning barriers in end-user systems [116], analyze the API usage of an IDE [38], understand developers' deprecation needs [241] understand how API documentation fails [276], evaluate the usability of the factory pattern in APIs [73], determine what makes APIs hard to learn [228], explore the pitfalls of unfamiliar APIs [69], and study API usability [211].

Empirical studies related to API evolution typically employ large data, case studies, or user studies to provide evidence of existing problems, the impacts of API evolution, or potential solutions to existing problems. These problems are a superset of the problems that are covered by the tools and solutions presented in this thesis.

Proposals and Reports

Proposals and reports within our dataset are papers that seek to highlight existing concerns in the field, and provide potential approaches to resolving the problem. These papers are categorized differently than other because they present a particular paper structure. These proposals and reports highlight issues that have been found in prior work (e.g., most API breaking changes are caused by refactorings [65]), and propose potential solutions to these problems (e.g., automatically detect API refactorings and replay them for clients [65]). However, these papers are proposals and do not provide complete solution details, nor do they evaluate the proposed solution.

Proposals and reports related to API evolution typically seek to highlight existing concerns in the field, and provide potential approaches to resolving these problems. We leverage some but not all of these concerns and findings as a basis for the approaches and tools presented in this paper.

Surveys

Surveys of existing literature seek to present a fair evaluation of a research topic by using a rigorous methods [129]. The surveys presented in this paper typically start with a research topic and observe existing literature to provide a view of the topic at hand. Our dataset contains five surveys related to API evolution.

In his 2016 survey on software ecosystems research, Manikas [159] seeks to provide updated evidence to determine and document evolution in the field of software ecosystems. The survey shows that the evolution of software ecosystems draws the attention of numerous papers [159].

As part of a book by Robillard [232], Mens and Lozano produced a chapter on Source Code-based Recommendation Systems [167], and Kim and Meng [232], on

Automating Repetitive Software Changes. These chapters can be independently obtained through the Springer archives, and we consider them to be two separate surveys of specific areas of recommendation systems because they are presented as such in *Recommendations Systems in Software Engineering*. Both of these chapters seek to provide state-of-the-art insight into specific recommendation Systems. Kim and Meng provide a general view of five source code-based recommendation systems and the in-depth design of one system to provide insight into the design decisions that are made when creating source-code based recommendation systems [232]. The chapter by Mens and Lozano seek to present state-of-the-art approaches that can be used to automate repetitive software changes [167].

In their survey of automated API property inference techniques, Robillard et al. [229], seek to provide an overview of API property inference techniques to present properties inferred, mining techniques, and empirical results of API property inference techniques [229].

In his survey on software merging, Mens [168], seeks to present a comprehensive analysis of available software merging approaches. The finding presented in this survey are directly applicable to API evolution topics such as API migration tools where merging techniques can be used to help automate API migration [165].

Survey papers, like this systematic literature review, typically seek to present an overview of a subject using existing literature to provide clarity for their given subject and allow for effective stepping-stones for future research. The survey papers we reviewed consider subject matters related to API evolution without concentrating on API evolution itself. We used these papers as a basis for some of our card sorting, as well as some of the terminology used in this thesis.

Datasets

Out of the 291 papers investigated for this study, we uncovered three papers that we labeled as dataset papers. Which concentrate on building a dataset related to some aspect of API evolution (e.g., Linux system calls [19]). The datasets are produced to conduct further studies [19], advance the state-of-the-art [3], and improve reproducibility of research [242].

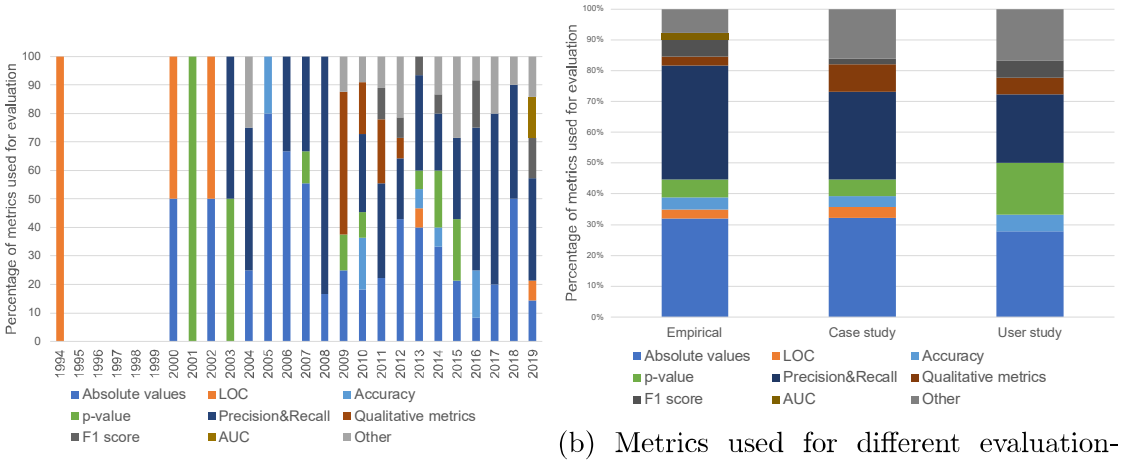
2.3.2 Evaluating API evolution experiments

We continue to answer our RQ1, to determine how API evolution research is typically evaluated. API evolution research often requires more than manually observing an API. Studies rely on distinct evaluation methods and make use of various Software metrics (e.g., precision and recall) to evaluate their results. Details for the various types of evaluations performed in API evolution are provided in Appendix A.

We identify four major means of evaluation used for API evolution research. Empirical evaluation, where quantitative metrics like LOC (lines of code) or precision and recall are used for evaluation over multiple subject systems. Case studies, where a single subject systems is used to obtain subject related metrics and results. User studies, that employ survey techniques and interviews with developers or users. Finally, qualitative evaluation that relies on subjective interpretations. Figure 7a presents the evolution trends of these four evaluation means. We concentrate on five paper types (i.e., New Tools and techniques, Empirical studies, Proposals and reports, Surveys, and Datasets) and identify the evaluation methods and the metrics that are used in these papers.

We identified 31 different evaluation metrics used in our publication sample. We assembled the metrics that occurred fewer than five times and were not known statistical properties (e.g., AUC, Confidence interval) into more global metric types, such as *absolute value metrics*, *qualitative metrics* and *other*. With those classifications we obtained 9 metric types. Their yearly trends can be found in Figure 6a.

Using the data we uncovered, we can see that although more rigorous evaluation metrics such as precision, recall, AUC, and F1 score appear to be gaining in popularity, a large percentage of papers still use a variety of non-standard absolute value metrics. A wide range of absolute value metrics are used to evaluate experiments and tools such as method parameter count, method changes, popularity, community size, project maturity, number of years active, fix rate, number of restarts [15, 92, 134]. None of these metrics are particularly flawed or bad, however the lack of standardization in the field makes it difficult to compare similar experiments and determine if progress is being made.



(a) Evolution of evaluation metrics

(b) Metrics used for different evaluation-types

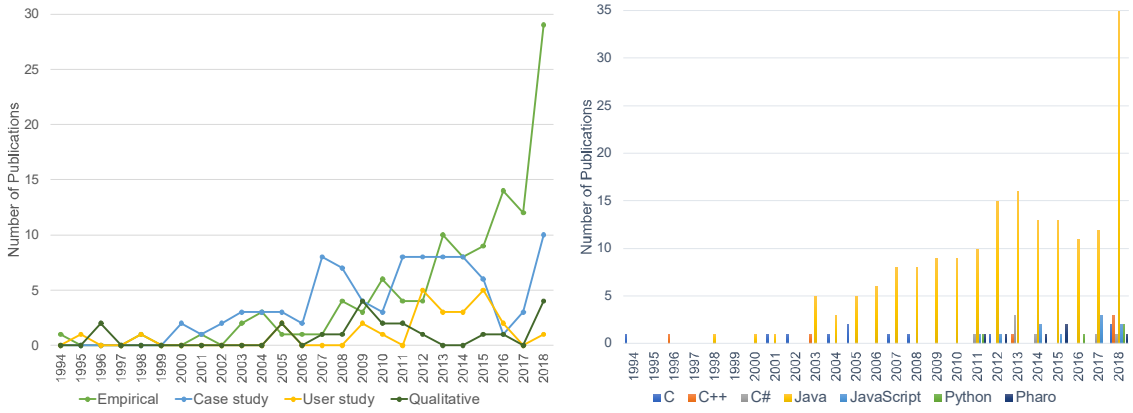
Figure 6: API evaluation metrics

New tools and techniques

As presented in Section 2.3.1 the majority of the papers fall within the scope of new tools and techniques. A surprising number of API evolution tools and techniques do not formally evaluate their tool, although this trend appears to be declining based on the last decade. In most of these cases, the tools appear to have been evaluated by the authors of the paper, however no formal evaluation is provided, e.g., when the tool is presented as part of a short paper, and is evaluated as part of a second paper. This is the case for the SemDiff tool by Dagenais et al. [51, 52].

Most API evolution tools are evaluated for their accuracy. In older papers, this accuracy was simply reporting the true positive rate [31, 163, 219, 260]. Recent papers reported precision, recall, F1-score, and the area-under-the-curve (i.e., AUC) [42, 112, 280, 282, 301]. Using Figure 6a we can see that in the last decade publications have begun to use more standardized metrics to evaluate their experiments.

In some cases it is not possible to ascertain the recall of a measure (e.g., in the case of mined framework usage changes [246]), then authors normally concentrate on providing precision metrics instead [50, 246, 259, 291, 303], which is particularly prevalent for data mined from large repositories for which it is impossible to manually determine if any instances were missed by the approach. It would be possible to remedy to this situation with high quality open-source datasets manually vetted by experts.



(a) Type of evaluation used over time (b) Language used for evaluation over time

Figure 7: API evaluation trends

Empirical studies

All of the empirical studies relied on some quantitative analysis to evaluate their results. The metrics evaluated depend on the API study, ranging from changes in APIs (e.g., addition, modification, removal) [146], changes in lines of code [162], code smells [96], API popularity [34], errors [180]. The most pervasive API evaluation criteria is absolute changes in API methods (e.g., changes to numbers of deprecated API methods, API methods added, API methods removed, API methods modified).

Case studies

As shown in Figure 6b, case studies present a variety of evaluations. Some case studies [134] compare absolute numbers of various metrics such as added APIs, deprecated APIs, removed APIs. Quantifying API changes through added/modified/removed APIs [76, 134, 227] appears to be a common evaluation methodology for API evolution case studies.

However, although most API case studies do consider and quantify API changes, some also rely on qualitative evaluations [22, 55, 226]. For example, one study [55] identifies six promises and seven perils of visualization tools, such as promising to provide feedback about errors. This qualitative information must be manually extracted by the authors throughout the case study.

Case studies appear to be well suited to uncover new evaluation metrics for APIs and to uncover previously unknown information, such as visualization tools [55], ripple effects caused by changes in software ecosystems [226], and API migration issues [22].

It is therefore expected that case studies present more uncommon absolute value metrics and other metrics, since these studies might be attempting to identify new metrics. The information uncovered through case studies can later be used in larger scale empirical studies of various APIs for example to determine the impact of API migration issues on various APIs [315].

Survey

We observe two types of survey papers related to API evolution. The first type concentrates on existing literature, for example Robillard et al. [229] surveys existing techniques and provides a summary of these techniques. Surveys of this type do not appear to rely on metrics to evaluate the papers presented in their findings. These papers instead rely on the evaluation presented in each of the papers surveyed. Furthermore, each survey of this type identifies a particular scope and specific criteria that must be respected throughout the study, criteria which are manually evaluated by the author(s). Similarly, in this systematic literature review, we also rely on the evaluations presented in our sampled papers. However, we also use quantitative information to uncover publishing and evaluation trends, as well as determine the emergence of API evolution sub-fields.

The second survey type provides the results of questions used to extract data from participants. These papers present quantifiable data that can be evaluated in various ways. For example, one paper [228] provides raw data for responses to survey questions within the related paper. The responses to the survey questions are quantitatively evaluated by the author [228]. Meanwhile, other works [73] survey the behavior of API users to specific tasks. This behavior can be quantified through statistical measures, such as standard deviation, Z-score, and p-values [73]. Current evaluation methodologies appear to be tailored to specific papers with no standardized dataset or evaluation methodology used for API evolution surveys. This lack of standardized evaluation methodology should be addressed by the community because it hampers comparison and makes it difficult to determine when and where progress has been made. Care was taken during this thesis to compare our approaches to existing tools and practices whenever possible. For example our migration tool, A3, was compared to an existing approach (LASE [165]) and our approach to aid with API misuse detection was compared using a standardized benchmark [3].

Proposals and reports

Reports from talks or expert panels of API evolution concentrate on coarse grained issues and challenges that plague the field of software APIs. These papers concentrate on abstracted problems taken from existing literature. Most papers that concentrate on future research avenues [27,123] and paradigm shifts [233] do not present evaluation criteria.

However, some exceptions exist. A report on Web APIs, concentrates on challenges in the field, but also suggests looking into metrics, like latency, to benchmark performance [290]. Similarly, papers on recommended practices concern specific software metrics that could be improved through developer knowledge (e.g., reducing coupling) [137]. Finally, a tool proposal contains an evaluation for the tool through accuracy metrics, and a user study [65].

Datasets

We found three papers that concentrate on presenting empirical datasets. Datasets related to API evolution, are proposed to stimulate research [242] and improve the state-of-the-art [3].

Datasets are not always fully evaluated because verifying large datasets requires a heavy manual cost. Therefore, some datasets do not present any immediate evaluation [242], some datasets are fully manually verified by multiple individuals [3], and some datasets are evaluated through manual verification of a statistically significant sample [19].

Evaluation in API evolution studies has not yet converged to specific styles and metrics. A surprising number of API evolution tools and techniques do not present a formal evaluation while some evaluate precision, recall, f1-score, and AUC. Meanwhile, API evolution empirical studies rely on various metrics with absolute changes in API methods appearing most often, but not always. Case studies, survey papers, proposals and reports, and dataset papers, all similarly present a variety of evaluation criterion.

Table 7: APIs used most commonly as evaluation subjects

API	Frequency
Java API	39
Android	30
Toy systems	20
Eclipse	16
JHotDraw	12
Log4j	11
Struts (1&2)	9
Guava	7
Hibernate	7

API	Frequency
JUnit	7
JFreeChart	7
Hadoop	6
Lucene	6
Pharo	6
Proprietary systems	6
Spring	6
.Net API	5

2.3.3 API subjects used for research evaluation

In this section, we finish answering our RQ1 by providing further insight into API evolution evaluation by presenting the APIs that are most commonly used as evaluation subjects. We concentrate on APIs used as evaluation systems in at least five different studies within our sample set. The frequency of API under evaluation in our sample set is presented in Table 7. While comparing the frequency of the various APIs used as evaluation subjects by prior work, we highlight benefits and reasons for choosing specific APIs as evaluation subjects.

We find that the majority (270) of the studies in our dataset employ at least one API to evaluate their hypotheses. 98 of these 270 studies employ multiple APIs to generalize results across multiple systems or multiple programming languages.

25 out of 291 publications either do not present or do not use an API. The lack of evaluation APIs may be due to the nature of the publication. For example, survey papers concentrate on summarizing the state of prior work [159, 167, 168, 229]. Similarly, book chapters [232], papers about general programming practices [137], future research proposals [70, 233], and hypotheses about the future of software engineering [123] do not employ APIs. Some tool papers do not provide any tests when presenting the tool [62, 65, 86, 190, 196, 256, 258, 299]. Similarly, exploratory research with theoretical findings does not always provide evaluations [1, 27, 290]. Finally, some research uses theoretical proofs to ascertain their results, and prove the validity of their approach without tests [40, 145, 236, 288, 293].

When considering that the Android API is primarily Java, and that most toy

systems (12/20) used within our sample are created using the Java programming language, we find that API evolution research is heavily skewed towards the Java programming language. As shown in Figure 7b, 190/270 papers are exclusively evaluated with Java systems between 1994 and 2019. The second most common programming language is C with 11 papers exclusively using C APIs to evaluate their findings. Figure 7b presents the evolution trends of programming languages used in the evaluation of API research. Furthermore, we only include programming languages that were used for more than two publications within our sample. Table 7 presents APIs that are used as test in more than five different research papers.

API evolution evaluation is heavily skewed towards the Java programming language, 190/270 papers that used API systems for some evaluation used exclusively Java APIs. Although Java is a popular programming language this representation is abnormal given recent programming trends. This presents opportunities for replication studies as well as potential avenues for future research with other programming languages which differ from Java in various ways, such as Python or Javascript. Unfortunately comparing new tools and approaches often requires a perpetuation of this problem. The work in this thesis unfortunately suffers from a Java bias because of this very issue.

2.4 State-of-the-Art in API Evolution Research

To answer our second research question (RQ2), we present the state-of-the-art in API evolution. We first present publication trends within the state-of-the-art in API research. We then concentrate on the most recent and seminal concepts and research in the field. We chose these seminal works based on the novelty of their content and the number of works that present similar ideas and build on these seminal works. We divide this section by publication contribution type as in Section 2.3.1. Table 8 presents a list of common API evolution challenges and state-of-the-art solutions proposed to resolve these challenges.

API Research State-of-the-art Publication Trends: As presented in Section 2.2.5 and as shown in Figure 1, we identified three primary API research topics, *API Maintenance*, *API Usability*, and *Other*. Figure 5a shows that both API usability and maintenance papers grew through the years. However, it appears that

Challenge	Proposed State-of-the-Art Solution
Dealing with breaking API changes	Document acceptable usages [289]
	API bundles [161]
	Automated API migration [36, 89]
	Extracting migration knowledge from clients [136, 245]
Improving API usability	Local interaction patterns [101]
	Usability patterns [321]
	API selection criteria [184]
	Digital assistants [27]
	Automated API tips [282]
	Automated documentation [2, 112, 151, 233, 278]
	Example mining [91, 182, 272]
	API recommendation algorithms [20, 152, 195, 304]
Reducing API misuses	Misuse detectors [4, 5, 16]

Table 8: State-of-the-art solutions to existing API evolution challenges

since 2011 the API research community has started to favor usability papers, with almost twice as many API usability than maintenance papers (31 vs. 17) in 2018.

Looking at the subtopics for API maintenance and usability in Figure 5, we can see that the *API Usage* research subtopic appears to be growing rapidly in recent years. This growth can likely be attributed to tools and empirical research to uncover what makes API hard to use [321], and uncovering usage patterns to help developers [289]. The growth in popularity for these topics might be linked to the growth in available API usage data on open-source repositories and forums such as GitHub and Stack Overflow, which were both launched in 2008. API migration research appears to be one of the more steady research subtopics with three to four publications per year since 2003. Meanwhile, the API misuses and recommendation subtopics appear to be gaining popularity in recent years. Although the first API misuse paper in our sample was published in 2001 [75], recent years have shown a steady stream of papers related to the topic, with three papers published in 2018 [5, 16, 223], and two in 2019 [4, 287]. The topic of API recommendation started gaining recognition in 2009 [212] and has been steadily gaining ground ever since.

2.4.1 Recent and seminal publications

Surveys

Surveys highlight seminal concepts and state-of-the-art work by design. As previously mentioned in Section 2.3.1, we found five survey papers pertaining to API evolution using the methodology highlighted in Section 2.2.

The survey papers highlight the state-of-the-art in recommendation systems that

pertain to API evolution [167,232], software ecosystems [159], API property inference techniques [229], and software merging techniques [168]. We use the metrics, classifications, and challenges uncovered by prior surveys to reinforce our own findings, in particular to categorize empirical studies as well as tools and techniques employed in API evolution studies into publication types in Sections 2.4 & 2.5.

The survey papers also highlight some open problems and future research directions in their respective domain. Some of the open-questions have been solved since the publication of the surveys. However, some challenges are still open, and we reiterate these along with our own findings in Section 2.5.

Surveys associated to API evolution tend to highlight the state-of-the-art in research as well as current research challenges and future research directions.

Empirical studies

We uncover two main subjects in the state-of-the-art API evolution empirical studies, those that concentrate on API usability and those that concentrate on API maintenance.

API usability

Many papers look into various aspects of API usability to reduce complexity [139]. These papers concentrate on issues such as breaking changes, integration problems, how API are used and what makes APIs hard to use, API standards, API misuses, and API documentation.

Current empirical studies in breaking API changes suggest that there is a growing need to document acceptable usages for APIs [289]. Furthermore, non-atomic refactoring patterns used by API developers can reduce API migration burdens [289]. Non-atomic refactorings in this case are defined as refactorings introducing a new API and changing the existing API piecemeal until there is no more use of the old API [289]. It is also possible to unbundle software APIs in different ways to vary the uniqueness of API bundles [161] allowing users to obtain bundles of features that suit their needs without breaking features that they would not use.

It has also be determine that, on average, dominant topics on forums can cover at least of 50% of questions pertaining to Web API integration [279].

Finding good names, relations between API types, knowing the impact of API

flexibility, and accurate documentation are all needed for good API usability [211]. API users claim that discovering allowable types is difficult, thus tools to suggest allowable types could benefit users [69]. APIs do present meaningful local interaction patterns that can be used for future recommendations [101]. Developers have a hard time understanding reflections API, and only produce tests after a bug is reported [214]. Developers use examples to understand how APIs work. They also need to understand the general idea of how an API works [228]

Recent papers have uncovered 22 patterns that determine what makes an API less usable [321]. Programming language [316] as well as tools, information, and boundary resources such as community are very important when selecting an API [184].

Issues pertaining to API standards [143, 179, 288] affect the usability of Web APIs [76]. Deprecation in particular has been found to vary mechanism, support, and implementation and fail to fully address the needs of developers [241]. Performance issues in mobile apps has been studied and carefully designing storage, limiting the MVC pattern, and limiting widgets are all factors that improve app performance [149].

Various works have studied API misuses [16, 124]. 11 different types of API fault cases have been identified [16]. Most cases have been attributed to missing data [16]. However, a lack of semantic awareness and correct usage examples lead to many false positives in API misuse detectors [5].

Many papers concentrate on API documentation motivated by incomplete documentation [74], the challenge of producing good documentation [206], and the shift of API documentation to more social sources [205]. A case study with Github and Stack Overflow to locate information from 10 popular APIs found that Github and Stack Overflow are often used by Google to document new functionalities [267]. An empirical study that combines API patterns extracted from GitHub projects to determine if Stack Overflow posts present faulty API code, found that up to 31% of posts may have potential API violations [309]. This indicates that users should be careful when using code from Stack Overflow. Languages with static typing and documentation are much easier to use than dynamic languages, with or without documentation [74]. This thesis mostly employs the Java programming language. The API issues presented in this thesis still arise even with a static programming language such as Java. If Java is to be considered much easier to use, this therefore indicates that there is still much work to be done for API usage in dynamic programming languages.

Documentation incompleteness and ambiguity plague developers in a user-study to determine what causes developers to use other APIs [276]. Almost all usage constraints are present in API source code but not in documentation [238]. An empirical study of automatic knowledge extraction techniques to extract knowledge from API documentation found that SVM and deep-learning methods can be complementary when attempting automatic knowledge extraction [87].

Papers on API usability typically pertain to challenges such as breaking changes, integration problems, how API are used and what makes APIs hard to use, API standards, API misuses, and API documentation.

API maintainability

A large number of empirical studies related to API evolution concentrate on the maintainability of APIs to conserve familiarity as APIs evolve [139]. More precisely, papers mainly concentrate on deprecation, reuse patterns, the speed at which APIs change, and the effects of propagating these changes.

A user study found that developers who use unstable Eclipse APIs often do not read documentation and therefore do not know which API are deprecated [38]. Empirical studies have been conducted to determine how effective documentation is at solving deprecation problems. Most documentation does not cover alternative APIs and code examples are very rarely documented [130]. However, in the case of the Android API, deprecated entities are removed in a timely manner, and the Android API recommends alternatives; yet most deprecated APIs in Android are in popular libraries [146] therefore users are still affected. Another empirical study determined that there is no major effort to update deprecation messages in most projects and that deprecated messages depend on the size and community of the project [35]. They found that only 64% of API elements that are deprecated have documented replacements, and that there is no effort to improve this over time [34]

Empirical studies have been conducted to detect reuse patterns and software clones to improve maintainability [120]. Patterns of API reuse have been identified in various code samples (e.g., opening and closing files) [170]. A decline of popularity appears to indicate that something is wrong with an API [172]. Some studies have shown that over 80% of breaking changes in API are due to refactoring [66], however other studies have since disputed this claim [48]. Refactoring APIs has however shown a

tendency to increase the speed at which bugs are fixed [126].

Empirical studies that concentrate on the side effects of rapid API evolution found that using new APIs that are highly touted may be a counter-productive practice [218]. Although the potential for problems to occur due to developers updating to newer library versions without modifying any of their source code is high, these problems tend not to occur on a wide scale in practice [61]. 28% of Android references are out of date. 22% of outdated API usages eventually upgrade to newer API versions but this takes about 14 months [162]. Mostafa et al. [180] found that most API incompatibilities are not well documented, and 67% of client bugs linked to backwards incompatibility can be fixed through simple client changes [180]. Furthermore, over 88% of Android apps follow the same workaround pattern to fix Android version issues, and this pattern can sometimes lead to incorrect behavior [98]. This pattern can be categorized as an instance of the multi-version workaround pattern presented in Chapter 6 of this thesis. Studies have suggested that developers believe there is a direct relationship between adopted APIs and user ratings [24]. Web services follow a spike and calm cycle of maintenance, an empirical study into Amazon services determined recommendations to make the most of spike and calm cycles from an API developer point of view [305].

API evolution empirical studies have been used to determine different patterns of evolution for Web APIs [144]. APIs change due to needing more functionality and usability [95]. Most API developers appear to introduce breaking changes to simplify the API and introduce new functionality [33]. Meanwhile, library maintainers are less likely to break API classes used by many clients [133]. API users rarely update API versions and only use deprecated entities less than 20% of the time. Most users do not react to deprecation, but will remove API references when something does get deleted from the API [244]. 14.78% of API changes break compatibility and impact 2.54% of clients [298]. Systems with higher break frequencies are usually larger and more popular [298]. Another empirical study similarly finds that about half of API changes cause reactions in only 5% of clients and that the overall reaction time is slow [107].

Studies have shown that mobile development questions increase when new versions of Android are released, and these questions appear to concentrate on deleted methods [150]. Meanwhile, mobile devs rarely update their apps, and when they do,

it is likely with respect to GUIs [239]. API updates are ignored due to poor awareness of benefits and high cost [239].

The results of these empirical studies lead to the recommendation of semantic versioning, self-documenting APIs, publishing customized change-logs with discussion forums for changes [249]. Furthermore, Web APIs should not change too often, old versions should not linger, API developers should keep usage data, blackout tests should be used, and providing examples is useful to users [78].

Papers on API maintenance typically concentrate on challenges such as deprecation, reuse patterns, the speed at which APIs change, and the effects of propagating these changes.

Proposals and reports

The proposals and reports primarily concentrate on highlighting an existing problem and proposing potential solutions for future work. API evolution proposals and reports concentrate on the future of API evolution research. The more recent proposals highlight the need to differentiate between Web APIs and library APIs [290] and to develop digital assistants to map user intent to ever more numerous APIs [27]. Furthermore, one particular proposal concentrates on a vision of automated developer documentation [233]. It highlights challenges such as establishing precise links between artifacts, capturing document request context, and the summarizing and synthesis of documents [233]. These proposals are particularly useful to understand the current demands of researchers and developers.

State-of-the-art proposals and reports related to API evolution concentrate on future research such as differentiating between Web and library APIs, automated documentation, and automatically linking software artifacts.

New tools and techniques

Over the years a variety of tools and techniques have been developed to ease the burdens caused by API evolution. In general we find that tools and techniques appear to primarily concern themselves with Lehmans law of *Conservation of Familiarity* while other laws such as *Continuing Change*, *Increasing Complexity*, and *Invariant Work Rate* serve as challenges to the *Conservation of Familiarity* [139]. We separate API

evolution tools and techniques into general topics such as *documentation* [53], *examples* [299], *misuse* [5], *migration* [66], *recommendation* [167], *usage* [229], and *other*. As presented in Section 2.2.5, these tool topics were either identified in prior surveys [159,167,168,229,232], or by using publication keywords, titles, abstracts as well as our own judgement. We provide a general overview of the state-of-the-art for each tool topic.

API documentation tools

API documentation has been described as large and cumbersome [58], lacking and difficult to produce [88], but instrumental to success [230]. State-of-the-art tools and techniques use Stack Overflow posts to supplement documentation and determine a match for lexical queries [112], augment documentation by automatically detecting APIs in the documentation [278], employ dynamic specification mining to improve decaying documentation [2], identify misuses in documentation to warn users [143], generate high quality source code summaries [151], and employ neural networks to produce high-quality text-to-code [188], code-to-text and code-to-code retrieval [194].

API examples tools

API examples have been touted as helpful to understand how APIs work [163,174, 176]. Approaches such as MAPO [318] and Jungloid [158] mine API examples from existing code. Approaches such as Examplore [91] employ relational topic models to produce API examples that span multiple files. Techniques using bytecode analysis [163], framework extension points [50], and software visualization [37] have also been used to identify API examples.

API misuse tools

API misuse tools primarily attempt to identify unfavorable API uses that could lead to future problems [4]. Approaches use machine learning [223], mutation analysis [287], specification mining [217], and API-usage-graphs [4], to attempt to detect misuses.

API migration tools

CatchUp! [62, 100] was one of the original approaches to deal with the problem of API migration. It captures API refactorings produced by API developers and synthesizes an edit script that can be replayed on API user code. Similar approaches were created where edit scripts could be manually created by the API developers [21] rather than recorded.

JDiff [12] is one of the first tools to synthesize a report of API changes between two versions of an application. It presents additions, removals, and modifications to any API. This information can be used to automatically track changes made to APIs. Similarly, ACUA [294] analyzes the binary code of both frameworks and client's programs written in Java to identify API changes, generating a report to estimate migration workload.

SemDiff [52] was one of the first approaches to use call dependency analysis to map APIs between two versions and determine a migration path between two or more migrated APIs. AURA [295] combines call dependency and text similarity analyses to identify API change rules between two versions. HiMa [166] uses revision control to create framework-evolution rules, which are then used to migrate user applications, outperforming both SemDiff and AURA. Approaches such as LASE [165] create a context-aware edit script from two or more examples and use the script to automatically identify edit locations and transform code. Recent API migration tools and techniques employ abstraction layers [89], knowledge extracted from API clients evolution [245], and syntactic changes [32] to improve API migration techniques.

Tools and techniques have also been proposed to migrate across programming languages rather than application version [36]. The state-of-the-art in this domain currently employs generative adversarial networks to produce high quality API mappings across languages such as Java and C# [36].

API recommendation tools

Identifying useful APIs can be a challenge for API users [186]. API recommendation attempt to ease the burden of selecting the most appropriate API by automatically recommending potentially useful API [221]. Various tools and techniques have been proposed to recommend useful API methods [42, 71, 212], and parameters [14, 308]. Current state-of-the-art approaches rely on converting English text queries and documentation to API elements [20, 195], ranking existing API recommendations by leveraging API usage path features [152], version history and Stack Overflow posts [17, 304].

API usage mining tools

Most of the tools and techniques are primarily targeted at API users. However, API usage mining tools are particularly suited to API researchers and API developers. These tools attempt to uncover various API usage metrics from API user projects

and examples. These tools and techniques are meant to determine API usage for a variety of reasons. These reasons range from determining the most useful API methods [256], to improving the productivity of API users [182]. We identified tools that automatically identify refactoring with high precision and recall [67, 272]. Tools that can automatically identify API that will be made public in the future [109], and tools that can extract fine-grained API usage [243].

Learning to use APIs appropriately is challenging [70]. Several attempts have been made at easing the learning curve of APIs by automatically improving online question/answer forums either through automatic answers [237], or by providing more information about the APIs themselves [210]. Other approaches use of machine learning approaches to extract and provide API tips to users [282]. There are also techniques that infer structured descriptions of Web APIs from Web examples [261].

Other API tools

Not all tools fit in the categories presented above. Some tools present solutions to niche problems to help verify the impact of APIs on program correctness [258, 284, 291, 292], software security [99, 114, 219, 319], and software quality [29]. We also found papers that detect deprecated APIs [320] and API reuse patterns or code clones [103,193,264,317] to identify useful patterns for API users and APIs to improve for API developers. Finally, tools have been created to apply standards to REST API [141], test cloud APIs [15], and develop adapters for Web services [28].

State-of-the-art tools related to API evolution seek to improve API usage, provide API recommendation, reduce API misuse, provide automated API migration, and better API documentation and examples.

Datasets

Papers that primarily concentrate on datasets are oriented towards replication, and future studies. In the three datasets in our study, the data presented is recent (2015-2018) and available online to be kept up to date and relevant to API evolution studies.

We identified a dataset constructed from the observation of a decade of Linux system calls [19]. This dataset presents 8,870 classified system call related changes. Another dataset presents 1,482,726 method invocations related to 5 Java APIs (Guava, Guice, Spring, Hibernate, EasyMock) created by mining 20,263 projects on GitHub [242]. Both of these datasets target research in software APIs to improve the state-of-the-art

in future API studies.

The final dataset specifically concentrates on API misuses [3]. This dataset contains 89 API misuses collected from 33 projects and a survey. The primary goal of the benchmark is to evaluate API-misuse detectors, which will then allow fair comparison between various approaches [3].

We only consider three papers that present datasets as primary contributions. However, papers listed under different primary contributions (e.g., Empirical studies) could have a dataset as secondary contributions. For example, there are papers that contribute approaches [243], or empirical studies [96] but also include datasets. Making research datasets open-source is becoming more popular.

State-of-the-art datasets are vetted, open-source sources of data that allow replications. Datasets of Linux system calls [19], API misuses [3], and API invocations [242] are all available for future research in API evolution.

2.5 Current and Future Challenges

Although API research has grown rapidly in the last decades, and several avenues of research have shown promising results and tools, there are still many unsolved challenges related to API evolution. To answer RQ3, namely *"What are the current and future challenges related to engineering APIs?"*, in this section, we identify existing API evolution research challenges and also uncover new ones. Table 9 presents an overview of the challenges that we found during this systematic literature review and for which we could not identify clear solutions. These challenges were scattered in the literature, which hides advances but also cloaks important, remaining challenges. We discuss each challenge in detail in this section. We number the challenges in Table 9 as existing challenges (*EC-1 through EC-17*) and challenges uncovered in this review (*CU-1 through CU-15*). We identify existing challenges for research on API evolution, new tools and techniques and empirical studies. We also uncover new challenges for API evolution, new tools and techniques and empirical studies, and API evolution datasets. We did not identify or uncover any particular challenges for proposal or surveys, nor did we identify any existing challenges for datasets during our review, therefore we omit these from this section. Based on our findings, we believe that Lehman's 8th law, namely *Feedback System* [140], poses the largest hurdle to future

Challenge type	Publication contribution type	Challenge
Existing challenges (EC)	New tools and techniques	*EC-1* Combining textual merging with syntactic and semantic approaches [168]
		EC-2 Providing a commercially viable API migration solution [32, 52]
		EC-3 Incorporating domain specific information into tools [168]
		EC-4 Using systematic evaluation methodologies in empirical evaluations [229]
		EC-5 Producing more specific and less abstract theories [159]
		EC-6 Reducing the variability of software API studies [159]
		EC-7 Finding input examples for API migration through examples [232]
		EC-8 Improving the granularity of API migration approaches [232]
		EC-9 Validation and correction of API migration edit scripts [232]
		EC-10 More tools to help with Web APIs [290]
		EC-11 Using existing library API research as stepping stones for Web APIs [290]
		EC-12 Combining both API side learning with client side learning [215]
		EC-13 Dealing with out-of-vocabulary problems [36]
	Empirical studies	*EC-14* Defining best fit APIs [307]
		EC-15 Automatically identifying factors that drive API changes [95, 111, 305]
		EC-16 Dealing with API semantics and dependencies [5]
		EC-17 Deploying bug fixes to multiple API versions [249]
Challenges uncovered in this review (CU)	New tools and techniques	*CU-1* Using uniform benchmarks for API tool evaluation
		CU-2 Reducing the context sensitive nature of API migration tools
		CU-3 Improving performance of API tooling to levels acceptable for user adoption
		CU-4 Dealing with fuzzy and ambiguous developer intent
		CU-5 Reducing the knowledge gap between API users and developers
		CU-6 Tools that mine usage data help API developers improve APIs
		CU-7 Keeping API users in the loop for API recommendation systems
		CU-8 Generalizing API tools to languages other than Java
		CU-9 Tools to help API developers deal with API migration, not just users
		CU-10 Reducing API misuse from the API development side
	Empirical studies	*CU-11* Understanding the coupling between API studies and programming languages
		CU-12 Determining the impact of API migration and the helpfulness of API recommendation systems across languages and API ecosystems
		CU-13 Generalizing API empirical studies to languages other than Java
		CU-14 Comparing the evolution of various APIs
	Datasets	*CU-15* Generate and verify large scale API migration and recommendation datasets

Table 9: Open challenges in API research

API evolution research.

2.5.1 Research on API evolution

Existing challenges:

In his survey on software merging [168], Mens highlights the need for tools that combine textual merging with syntactic and semantic approaches (*EC-1*). This was attempted in API migration tools like SemDiff [52] and APIDiff [32]. However, these tools have yet to provide a widely available commercially viable solution (*EC-2*). Mens further highlights the need to incorporate domain specific information, which has also been attempted by various API migration tools, with various levels of success (*EC-3*). However, current solutions appear context sensitive.

Robillard et al. [229] found that the empirical evaluation of API properties is lacking in systematic evaluation methodology (*EC-4*). Although their survey determines a foundation to compare API property inference techniques, this methodology has yet to rise. It is unclear why this foundation has yet to take hold. Perhaps due to

a lack of exposure, or because there are hurdles imposed by the proposed systematic evaluation methodology. We hope to bring attention to this challenge amongst others, to improve the exposure of existing proposed evaluation methodologies, and guide future research into more systematic and comparable evaluations. In particular in this thesis we use existing tools (in Chapter 4) and benchmarks (in Chapter 5) to evaluate our results.

Manikas et al. [159] posit that theories about software ecosystems and the APIs they involve can often be either too general (*EC-5*) or too abstract. Manikas highlights that it is difficult to study software ecosystems due to the high variability in the field, APIs which are part of these ecosystems are therefore similarly impacted by high variability (*EC-6*).

Robillard et al. [232] highlight several open challenges with respect to automating repetitive software changes. Finding input examples to automate software changes remains an open problem (*EC-7*). Integrating testing with code recommendation and dealing with various levels of code granularity (*EC-8*) for API recommendations and migrations also remain open challenges. Current recommendation tools rely on human intervention to determine the correctness of the recommendation (*EC-9*). Tools such as MAPO [318] attempted to automate API example gathering, but no tool currently fully solved this challenge. Work remains to extract code examples relevant to user queries, and to determine whether multiple examples are similar.

Uncovered challenges:

Currently, API property inference techniques do not appear to use uniform benchmarks to test their performance. The results of these techniques are therefore at the mercy of the dataset and evaluation methodologies chosen by their authors which prevents comparisons between techniques. Future research should seek to use a standard evaluation such as the one provided by Robillard et al. [229] to improve the ease of comparison between various approaches (*CU-1*). Current solutions appear context sensitive, and it has been proposed to incorporate domain specific information into tools to remedy to this problem. Yet, it is unclear how to reduce the context sensitive nature of API migration tools and how these approaches would perform on different datasets (*CU-2*) or how their usage might affect API evolution feedback loops.

We posit that although there are some studies that attempt to generate theories about software APIs [101, 149, 211, 321], most tools and studies appear to be either

dependent on, or linked to, factors such as API ecosystems and programming languages of the API (*CU-11*). Few studies attempt to determine whether the severity of various API evolution problems such as API migration and API recommendation are present across all programming languages (*CU-12*). Systematic studies to determine the impact of API migration and the helpfulness of API recommendation systems are required to understand whether such aid is universally required or language dependent.

2.5.2 New tools and techniques

Existing challenges:

Most of the tools presented in this report concentrate on library APIs, and very little effort has been done on Web APIs [290] (*EC-10*). Web API users must concern themselves with quality of service, weak specifications, and a lack of comprehensive listings for Web APIs [290]. Web APIs similarly suffer from API migration, API documentation, and API example problems. Researchers should therefore use existing research, such as existing API migration approaches [12,32,36,52,89,165,245,294,295], high quality code summary generation [151], misuse identification [143], and using relational topic models for examples [91] as stepping stones to improve Web API tooling (*EC-11*).

State-of-the-art migration techniques should consider hybrid approaches (*EC-12*) to combine both API side learning with client side learning [245] and consider the use of domain adaptation methods (*EC-13*) to deal with out-of-vocabulary problems [36].

API migration, API recommendation, and API misuse detectors still have room for improvement. These challenges require keeping the API users in the loop, because they are ultimately the ones most impacted by these problems. Furthermore, tools that attempt to aid with these problems should aim to support more programming languages and Web APIs.

Uncovered challenges:

Many tools and techniques have been created to deal with API evolution challenges. However, most tools concentrate on a small range of challenges and do not fully consider feedback loops involved in API evolution. Although individual tools show promising results, none can claim to be 100% effective at solving their target problem. It remains unclear whether current approaches are good enough for user

adoption, or if performance should still be improved before users can start using these tools (*CU-3*). Fuzzy and ambiguous intent (*CU-4*) as well as the rapid evolution of software services that employ APIs, such as IoT devices, are challenges that concern evolving APIs [27]. Effective API engineering must find solutions to deal with technical problems caused by APIs, and to reduce ambiguity of APIs and the knowledge gap between API developers and users (*CU-5*). New tools are needed to help API developers create APIs that are easy to use by API users (*CU-6*), just like better techniques are required to help API users understand how to use APIs (*CU-7*). Both of these challenges are dependent on researchers understanding what constitutes a "good" API, and why API users select one API over another.

Many tools want to expand to more programming languages [108,112,182,282,303]. However, most are still developed for Java. Figure 7b shows no discernible shift to other programming languages in recent years. Therefore, it remains to be seen how effectively API evolution tools would translate to other programming languages (*CU-8*).

API migration received a great deal of attention in API evolution research. However, it is still an open problem. Most existing approaches concentrate on the client side, with the premise that API migration is the burden of API users. Little research has been done to determine if it would be more efficient to transfer some of the burden to API developers (*CU-9*), and then develop tools to improve API engineering such that API migration efforts are reduced on the client side.

Several tools have been developed to extract API misuses and API usage (e.g., API call frequency). Little research concentrated on using usage and misuse information to create a feedback loop to help API developers improve their APIs (*CU-10*). Most of the API research conducted in the last two decades concentrated on API users rather than API developers.

2.5.3 Empirical studies

Existing challenges:

Various empirical studies uncovered the need for future studies on API developers and API development for supporting the evolution of APIs [77, 226, 227], defining best fit APIs [307] (*EC-14*), and automatically identifying factors that drive API changes [95, 111, 305] (*EC-15*).

In their study on API misuse detectors, Amann et al. [5] highlight the need for future studies into program semantics and dependencies (*EC-16*), as well the need for tools that properly handle alternative patterns for the same API.

The need for tools to deploy bug fixes to several versions of an API at once (*EC-17*) has been proposed by Sohan et al. [249].

Uncovered challenges:

Most (66%) API evolution empirical studies concentrate on APIs written in the Java programming language. Other languages such as C, C++, C#, JavaScript, Python are only covered by a small percentage ($\leq$ 5% each) of empirical studies. Future studies should generalize to languages other than Java (*CU-13*).

A great number (74%) of empirical studies do not rely on any statistical tests to evaluate their results. The majority of these studies present metrics such as lines-of-code (LOC) or the numbers of field/method/class changes, but there is no current way to normalize these results to compare them across studies or APIs (*CU-14*). It remains an open challenge to compare the evolution of various APIs, particularly across programming languages.

2.5.4 Datasets

Uncovered challenges:

We identified three papers on datasets. Although it has become more popular in recent years to publish datasets (all four datasets presented in this paper were published after 2015), the field suffers from a lack of accepted and up-to-date datasets. For example, 13 papers concentrate on API migration tools and techniques, however, we could not identify any common dataset or API to directly compare migration tools or studies.

The field of API evolution would greatly benefit from more datasets, particularly with respect to API migration and API recommendation (*CU-15*). However, it remains challenging to generate and particularly to verify these datasets, because some API migrations and recommendations can be subjective and context sensitive.

2.6 Chapter Summary

In this chapter, we presented a survey of existing API evolution literature. We uncover the publication trends as well as common questions and goals of research papers related to API evolution. We find that there are five general types of research papers within the scope of this paper. We find and highlight various state-of-the-art approaches and findings within sub-areas of API evolution such as API usability, API maintenance, API migration, API recommendation, API misuses, and API documentation. Furthermore, we also present various methods and popular subject APIs used to evaluate API evolution research. We highlight some drawbacks of existing evaluation methods and present some potential future challenges that could be undertaken by future research. We also highlight important, remaining challenges within the scope of API evolution, and present some novel challenges that were uncovered during this literature review. Although we find that *continuing change increasing complexity*, *conservation of familiarity*, *continuing growth*, and *declining quality* all pose worthy challenges to API evolution, we believe that the next hurdle will be leveraging and mastering the , *feedback systems* involved in API evolution [140]. In this thesis we contribute to API feedback systems by leveraging API user and API developer data and feeding it back to them in contextually relevant situations such as during API migration, and by reporting interesting API workarounds as potential API improvements to API developers. We hope that this chapter can act as a reference for existing work within the scope of API evolution, as well as present challenges to guide the future of API evolution research. Furthermore, we used the information presented in this chapter to shape the overall direction of this thesis. Although we do not attempt to solve all of the challenges presented in this section, we do attempt to solve challenges that were presented in this section. We also rely on existing API evolution research that was presented in this chapter as a basis for our work.

Part II

Aiding API Users

Chapter 3

What are the Challenges Associated with API Migration?

In recent years, open-source software libraries have allowed developers to build robust applications by consuming freely available application program interfaces (API). Studies on API migration often assume that software documentation lacks explicit information for migration guidance and is impractical for API consumers, however the Android API appears to go against this trend. Past research has shown that it is possible to present migration suggestions based on historical code-change information. On the other hand, research approaches that rely on the existence of API documentation have also observed positive results. Yet, the assumptions made by prior approaches have not been evaluated on large scale practical systems, leading to a need to affirm their validity. This chapter reports our recent practical experience migrating the use of Android APIs in FDroid apps when leveraging approaches based on documentation and historical code changes. Our experiences suggest that migration through historical code-changes presents various challenges and that API documentation is undervalued. In particular, the majority of migrations from removed or deprecated Android APIs to newly added APIs can be suggested by a simple keyword search in the documentation. More importantly, during our practice, we experienced that the challenges of API migration lie beyond migration suggestions, in aspects that require knowing how to migrate, such as coping with parameter type changes in new API.

3.1 Introduction

The current trends of mobile computing and software as a service present an increasing need for developers to rely on externally maintained software rather than consume their valuable development time [148]. However, as a consequence, software developers become dependent on frameworks and public application program interfaces (APIs) when developing their applications [48,175,186]. When programming with an API, consumers must either use available documentation or code examples, in order to guide them in consuming the targeted API [162,186].

As of 2016, the Google Play application store presents over 2.2 million applications [240]. All of these applications rely on the Android API to access device information and drivers. Released in September 2008 [178], the Android API is currently in its 28^{th} version. This API provides a large number of varied functionalities for its consumers exposing more than 19,000 public methods. With over 1.5 million daily activations of Android devices, the use of the Android API is expected to keep growing in the coming decade [252].

Because the development of the API is typically independent from the consumption of the API, the consumers are at the mercy of the evolution of the API. Finding ways to ease the API migration burden by introducing documentation and guides to help with API migrations, or API migration pathways, is therefore a boon to API users. Prior research has concentrated on recommending or producing specialized tools to provide suggestions for consumers pursuing API migrations [14, 48, 51, 66, 198, 201, 253]. These tools use various inputs, such as code documentation [12,253] and historical code-change information when producing API migration suggestions [51,63,100]. Yet there exists no large-scale study to assess the usefulness of these approaches in real-world API migrations. The popularity and the importance of the Android API makes it an ideal subject to conduct such a study.

In this chapter we report our experiences with Android API migration using strategies described by prior research, namely those based on documentation and historical code-changes. Our findings are summarized in Tables 10 & 11.

As a first step, we opt to leverage the Android documentation [9], due to the important role of documentation claimed by prior research [12, 18, 177, 205, 233, 268, 301]. We find that although not all migrated methods can be found in the official Android online documentation, information needed to assist in API migration can be

Leveraging documentation (Section 3.3.1)	**Implications**
1) For 26% of the deleted or deprecated Android API methods, we could not find any replacements in our manual examination of the API.	Developers of migration tools and API consumers should be aware that not all modified methods have migration pathways. This might mean supporting an old version of the API or changing functionality.
2) Android documentation, including the online documentation, code comments, and commit messages often contains useful textual information for method migrations as well as information for their deprecation, addition, and removal per API version.	Android app developers should leverage such effective documentation as it allows them to understand and plan their API uses and migrations around the modifications to the API.
3) Migration pathways in documentation are often explicit. Links between methods are declared, and replacements are identified with method signatures for easy recognition.	Android app developers could recover Android API migration links by simple keyword search, instead of exploiting sophisticated techniques.
4) The Android official documentation [9] effectively presents migration pathways. Based on documentation alone, with naive text matching, we were able to automatically determine most of our manually identified migration paths.	Due to the high quality and the ease of access of the Android documentation, suggesting Android API migrations may not be a challenging task. Instead, migration research should concentrate on other tasks, such as handling different migration types.

Table 10: Findings and implications on Android API migrations

also found in other forms of documentation, such as code commit messages and code comments. Still, the official documentation contributes the majority (75.3%) of the information to suggest API migrations.

In a second step, we leverage historical code change information (e.g., commits) to improve the results of the API migration suggestions from the previous step. In particular, prior approaches [51, 225, 253] that are based on this information typically assume that API migration information can be found in the source-code commits, i.e., if a method is removed, a replacement method should be added promptly, and can therefore be found in historically close code commits. Therefore, we first examined this assumption in order to understand whether the techniques that are proposed in

Leveraging historical code changes (Section 3.3.2)	**Implications**
5) Historical code data, such as commits, only yields a few undocumented migration pathways and a fraction of migration pathways contained in documentation.	API migration researchers should employ historical code data as a backup when documentation is lacking, and not as a primary migration pathway source.
6) In the Android API, replacement methods can be introduced earlier or later than the existing method, with a large time gap. This breaks some history-based automated API migration assumptions.	Android app developers should verify the assumptions of automated migration tools before exploiting them in practice.
API migrations in FDroid apps (Section 3.3.3)	**Implications**
7) Actual modified API usage centres around a few API calls. Most API users only require support for few modified API methods.	Android app developers and Android API architects could mine API usage data to prioritize their migration efforts.
8) API migrations often require further code modifications than simple renames or parameter changes, e.g. object instantiation.	Future research on API migration should investigate automated support to suggest code modifications examples for API migration.

Table 11: Findings and implications on Android API migrations cont.

prior research can be leveraged in migrating APIs in practice. We found that most of removed/deprecated methods and newly introduced API methods for migration do not change in the same code commit. 30.4% of the new Android API methods are not even introduced in the same version as the removed/deprecated Android API methods. Furthermore, historical code change information only provides 42.7% of the necessary migration suggestions, and 90.5% of those are already indicated in the documentation.

To test the effectiveness of identified migration pathways, in a third step, we leveraged the API migration suggestions that we automatically recovered from both documentation (including official online documentation, commit message, and code

comments) and historical code change information for some FDroid apps.[1] We experience that only a small subset of the removed/deprecated API methods and API migrations are used by FDroid apps.

Our results and experiences imply that even though documentation is often reported as incomplete or outdated [51], developers should still consider the official documentation of the Android API as their major source of information. Moreover, before using any sophisticated techniques for API migration, developers should first verify the assumptions of those techniques before exploiting them in practice. On the other hand, developers could reduce and prioritize their efforts to a small subset of API methods, which are used in practice.

Our experience agrees with prior research [12,51,100,198,253,259] and shows that it is feasible to provide suggestions when migrating API methods to new versions. However, more importantly, after we successfully performed API migrations on three apps from one version of Android to the next, we found that implementing API migration code changes is much more challenging than identifying what modification should be done to migrate from one API to another. Challenges such as migrating multiple related-APIs as well as changing object types present changes that would often require extensive knowledge and effort. We document these challenges in this experience report so that further research on API migration can investigate and propose automated solutions to assist API migration in practice. In this thesis we provide one such tool in Chapter 4.

The contributions of this chapter include:

- We evaluate the use of documentation and historical code change information in API migration in a large scale subject.
- We find that the information needed to identify replacement API methods for migrations often resides explicitly in online documentation and repository commits as natural language text.
- We find that prior research-based sophisticated migration techniques may fail because particular assumptions are not met in practice.

[1]Our automated script to recover Android API migration pathways is hosted online at https://github.com/LamotheMax/MSR_2018_Android_API_Study

- We documented our experiences and the challenges that we encountered when migrating the use of Android APIs in FDroid apps to benefit both practitioners and researchers.

- We documented the solutions we employed for our challenges, and presented our unsolved challenges as open challenges for future research.

The rest of the chapter is organized as follows: Section 3.2 provides a background on API migration practices and past research. Section 3.3 presents the methodology followed in our study and reports our experiences the challenges we encountered. Finally, Sections 3.4 and 3.5 outline threats to validity and a summary of this chapter.

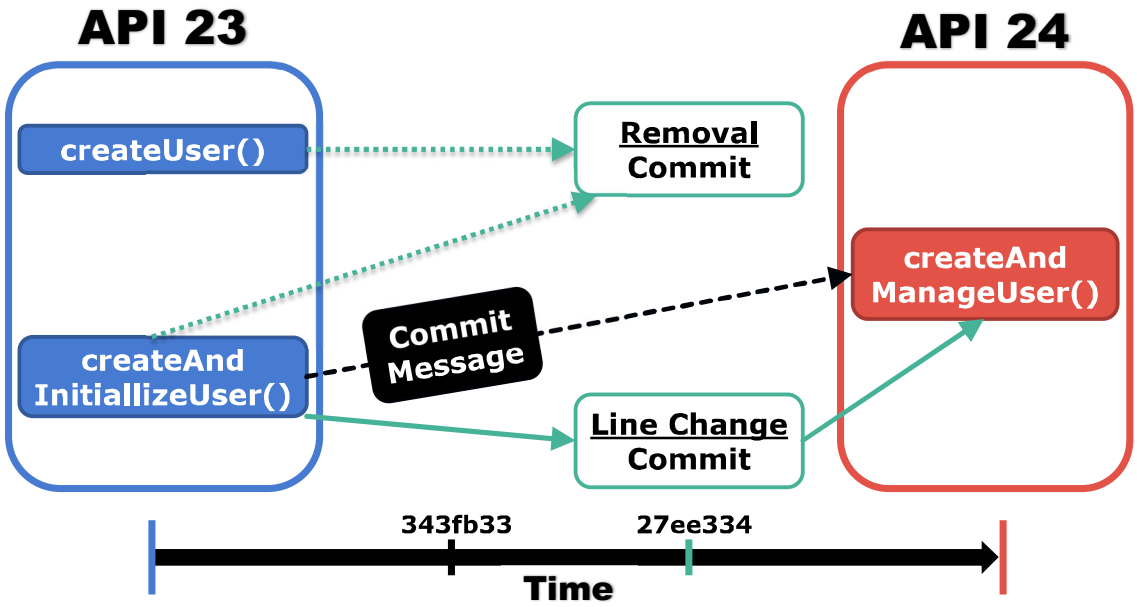
3.2 Android API Migration

Android app development heavily depends on the availability of Android APIs. In the most recent version of Android, "Oreo", there currently exist 3,354 API classes and 33,560 API methods. However, such APIs are updated every 6 to 12 months when Android releases a new version. A prior study by McDonnell et al. showed that on average 115 Android APIs are updated per month [162]. Such API updates would cause around 28% of API references to become outdated in a median lagging time of 16 months, while upgrading these updated APIs takes about 14 months [162]. Prior research shows that Android app developers seek to update their API usage, however their migrations are slower than the API updates [162]. Therefore, efficient migration techniques are essential to help Android app developers.

Android architects maintain an open online documentation to share information on available API and to communicate API deprecations in each version [9]. However, knowing that an API is deprecated may not give the consumer enough information on the existence of a replacement API or which new API is needed to replace any loss in functionality.

For example, the Android API has been releasing new versions since September 23, 2008 [178]. The Android project provides a number of resources to help consumers keep track of changes in the API. However, even with its well maintained documentation, it is sometimes required for an API consumer to look at the API source code to determine how to migrate a removed API method as presented in Section 3.2.1.

Figure 8: Example of methods that are linked through commit history.



In the Android project, behaviour changes of API methods are sometimes documented and presented with new API releases. This documentation can be used to locate method substitutions directly, and this has been used in this research in order to check the results of our links. However, not all versions of the API provide this documentation. We assume this documentation is rarely done due to the resource requirements of maintaining such a list. Having a tool to do this automatically, or at the very least to check the list for errors, would be a welcome boon for maintenance efforts.

3.2.1 A real-life example

API Documentation alone is sometimes insufficient to determine the migration of a removed method. For instance, the API method `createAndInitiallizeUser` from `android.os.UserManager` was removed between Android API versions 23 and 24. The method was replaced by a new method, `createAndManageUser`. Since the method was removed, no information on the methods is henceforth available in the most recent Android official online documentation. However, it is possible to find the removal of the method in the change documentation of the API [9].

In order to determine the genuine evolution of the method, one needs to look

at the revision history of the framework. Due to the open source nature of the Android framework, the API version control repository is available online. Official online documentation of the `createAndInitiallizeUser` method does not provide useful information for the migration. However, by looking at repository commits, it is possible to see that the `createAndInitiallizeUser` method was replaced by the `createAndManageUser` method in commit *343fb33* as shown in Figure 8. The information can also be found in the internal code comments and commit messages. Moreover, by looking at other commits during the history of the Android version, we find another method `createUser` that is co-changed with `createAndInitiallizeUser`, while having no other documentation about the removal. This requires creativity and research on the part of the user in order to find a substitute. However, by looking at the commit history, and carefully parsing the commit comments, it is possible to determine that the `createAndInitiallizeUser` and `createUser` are interlinked through removals, modifications, comments, and Java class, but they are never explicitly linked. Therefore, a user that wishes to update their use of the `createUser` method should also take a careful look at the `createAndInitiallizeUser`.

This example is particularly interesting because it shows that:

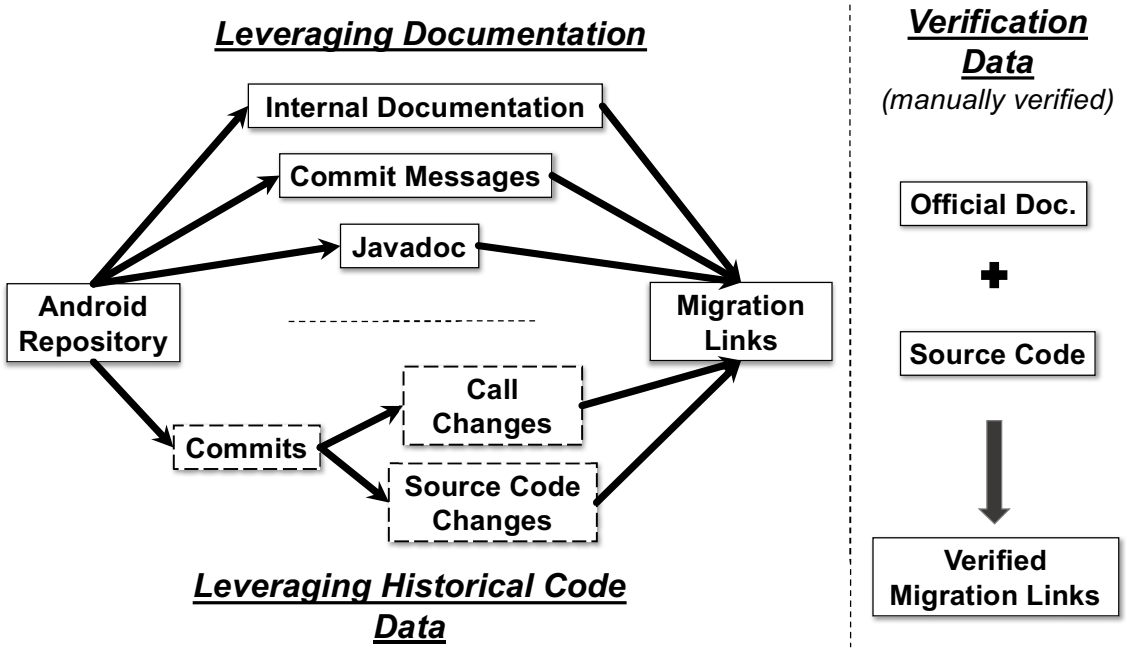
- Not all methods that have replacement methods present the information in official documentation.
- Consumers of methods such as these are expected to put in the effort to find the replacement themselves.
- API migrations can involve multiple API methods.

Examples such as these are the primary motivations for this work. Ideally, with a complete mapping of all methods and their relationships to other methods, developers should be able to get an understanding of migrations with a simple glance. Therefore, in this paper, we aim to evaluate the applicability and usefulness of existing approaches on automated Android API migration.

3.3 An Experience Report

In this section, we aim to explore the use of existing automated API upgrading techniques to migrate the Android API. In particular, we explore the use of documentation and historical code change information based techniques to assist in migrating removed or deprecated Android API methods. In order to evaluate the use of existing automated API migration techniques, we first need to extract all the changes to Android API including additions, deletions, deprecations, or modifications to existing API in each version of Android. In particular, we select the most recent six versions of Android (21 to 26). We first leverage JDiff to identify all added, removed, deprecated and modified APIs between every two Android versions. The Android API changes are summarized in Table 12. In particular, we consider the total amount of removed or deprecated API as the upper bound of all possible API migrations, since they would suggest or even force developers to change their source code in order to adapt to new APIs.

Figure 9: API migration extraction strategies.



In the rest of this section, we report our experiences during the migration of APIs uses in Android apps. We discuss our experiences in three steps. For each step, we discuss its motivation, our approach, and the outcome of the step as well as the

Table 12: Android API modifications per API version

		Class			Method			Field		
API Version	**Release Date**	Added	Changed	Removed	Added	Changed	Removed	Added	Changed	Removed
16	2012-07-09	57	211	0	381	151	20	171	46	4
17	2012-11-13	41	111	2	150	37	19	155	69	3
18	2013-07-24	61	108	36	155	44	4	131	25	1
19	2013-10-31	78	180	0	268	26	5	391	6	0
20	2014-06-25	10	25	0	32	5	1	45	2	0
21	2014-11-12	147	360	0	770	117	29	1150	75	2
22	2015-03-09	4	439	0	73	128	3	53	1	13
23	2015-10-05	119	257	36	541	132	38	466	89	83
24	2016-08-22	147	133	3	877	127	13	585	31	5
25	2016-10-04	5	38	0	50	1	0	53	0	0
26	2017-08-21	145	349	4	795	139	18	572	69	0
Min		4	25	0	32	1	0	45	0	0
Max		147	439	36	877	151	38	1150	89	83
Mean		**57**	**155**	**4**	**266**	**58**	**10**	**254**	**35**	**8**

For the purposes of this study we concentrated on the six most recent versions (API Versions: 21-26)

challenges that we faced.

3.3.1 Step 1: Leveraging documentation in API migrations

Motivation

Due to the high dependence between Android apps and Android APIs, ideally, all removed and deprecated APIs should be properly documented, such that consumers can opt to adopt other APIs to sustain the functionality of their apps. In addition, previous research has shown that documentation can be used to determine migration pathways in changing API [253]. Therefore, in this step, we seek to determine whether the Android API documentation can be leveraged when assisting with API migrations.

Approach

In order to automatically recover API migration suggestions from documentation, we consider three readily available sources of data as documentation: 1) code comments in JavaDoc format in the source code, before the declaration of each method, 2) code commit messages and 3) official Android online API documentation.

Code comments in JavaDoc. We first obtain all the source code for each version of Android. We then use srcML [46] coupled with python scripts to extract all the JavaDoc code comment for each Android API. For each code comment, we use the API name as keywords, and automatically search whether the name of a changed (added, deleted, or deprecated) API is mentioned in the comment. For

example, as shown in Figure 10, in API version 23, `android.content.res .Resources.getColor(int)` was deprecated and obtained a JavaDoc link to its migrated method: `android.content .res.Resources.getColor(int, Theme)`.

Figure 10: `getColor(int)` source code snippet presents a migration pathway.

```
* @return A single color value in the form 0xAARRGGBB.
* @deprecated Use {@link #getColor(int, Theme)} instead.
* /
@ColorInt
@Deprecated
public int getColor(@ColorRes int id) throws NotFoundException {
        return getColor(id, null);
}
```

Code commit messages. We extract all code commits and their commit messages between every two consecutive versions of Android from the *git* repository [10]. Similarly to code comments, we automatically search whether the name of a changed API is mentioned in the code commit message. For example, as presented in section 3.2.1, and in Figure 11, `createAndInitiallizeUser` presented a link to `createAndManageUser` in commit message 343fb33.

Official Android API documentation. The Android API documentation contains a list of added, deleted, or deprecated APIs in each version [9]. By checking the online documentation of each deleted or deprecated API, we manually examine whether the official documentation provides a replacement for the deleted or deprecated APIs. For example, `android.text.Html.fromHtml(String)` appears in the online documentation as shown in Figure 12. It is also possible to mine the documentation from JavaDoc links in the historical code-data information. The `createAndInitiallizeUser` method and its migration also appeared in online documentation, however only in the framework repository documentation [10].

Results

The majority of replacements for deleted or deprecated APIs can be recovered from the explicit wording in documentation. For the six studied versions, we were able to determine between 51% and 98.4% of the deleted or deprecated APIs through documented replacements. We then manually examined the APIs for which we could not recover a replacement, in order to understand whether

Figure 11: Android framework commit message 343fb33, presents a migration pathway.

> **Add new API function createAndManageUser**
>
> This is a reduced version of the (deprecated) function **createAndInitializeUser**, that allows the device owner to create a new user and pass a bundle with information for initialization. The new version of the function has the same functionality, but the profile owner of the new user is always the device owner.
>
> A flag can be specified to skip the setup wizard for the new user.
>
> The new user is not started in the background, as opposed to how **createAndInitializeUser** did it. Instead, the bundle with initialization information is stored and will be broadcast when the user is started for the first time.
>
> Bug: 25288732, 25860170
> Change-Id: I4e1aea6d2b7821b412c131e88454dff5934192aa

those APIs do not have a replacement or whether we missed a documented replacement. For 26% of the deleted or deprecated APIs, we cannot find a replacement at all (possible removal of functionality). For example, all of the methods present in the `PskKeyManager` class were removed without replacement when the class was removed due to incompatibilities with TLS 1.3.

Experience #1: For 26% of the deleted or deprecated Android API methods, we could not find any replacements in our manual examination of the API.

We also found an extra 21.5% of replacement APIs which exist, but we were unable to recover them explicitly from the documentation. Some knowledge of the project had to be combined with the documentation. For example, if method functionality had been migrated to a different class, which existed prior to the studied release, we were unable to provide a replacement automatically. This was particularly prevalent when methods migrated from using static methods to classes which

Figure 12: Android online documentation for method `fromHtml`, presents a migration pathway.

produced similar results through new objects. Externally maintained replacements as part of the `java.lang.Math` package account for 8 of the 29 replacements which were not found for API version 23. One of the undiscovered replacements was a Java wrapped C method which proved difficult to link. However, we were able to link a similar method in API version 24 through its documentation links. Out of the 15 other undiscovered replacements, all of them referred to another class to replace the lost functionality.

Experience #2: Android documentation, including the online documentation, code comments, and commit messages often contains useful textual information for method migrations as well as information for their deprecation, addition, and removal per API version.

Experience #3: Migration pathways of APIs in documentation are often very explicit. Links between methods are clearly stated in the documentation, and replacements are identified with complete method signatures for easy recognition.

The official documentation of Android API is the main source of data for suggesting API migrations. Android provides a rich documentation from the official documentation website [9], and from the framework commits [10]. We find that the documentation of Android API provides more migration links than any other sources. By manually inspecting all the sources of information of the removed or deprecated APIs, we find that only 5 out of our 469 studied APIs had migration paths that were not presented in the official online documentation. However, as presented in

challenge #2, not all previous documentation is readily available in the latest online index [9], and some of it must be mined from the JavaDoc in the repository [10].

Our results show that identifying replacements for the removed or deprecated Android APIs may not compose a challenging task, since developers may not need sophisticated techniques to analyze documentation in order to detect the API replacement, while simple keyword searching on the API names may recover the majority of the API replacement. More importantly, the majority of the replacements can be recovered from the official documentation. Compared with the code comments and the commit messages, the online Android official documentation is the easiest to access and to analyze by consumers. This finding also implies that developers may not need sophisticated techniques nor access to the software repositories to migrate Android APIs.

Experience #4: The Android official documentation [9] effectively presents migration pathways. Based on documentation alone, with simple text matching, we were able to automatically determine most of our manually identified migration paths.

Challenges

Challenge #1: Associating documentation to APIs.

Description: Source code documentation is not always favourably located in order to determine the targeted source code artifacts. During this research we noticed that sometimes documentation provided at the top of a Java class can give migration or removal information about a method in the class. However, linking this documentation requires foresight of its existence.

Our solution: Since our method of using naive text matching worked well for migration suggestions, we determined that in the case of the Android API we could also apply text matching to documentation found throughout a given class (cf. Table 13). After identifying the removal of a method in a given API we suggest looking at all unlabelled documentation in its class for text matches, and attempting to identify any other method mentioned.

Challenge #2: Missing historical information of API documentation.

Description: All references to removed methods are expunged in the official Android documentation. Therefore, when a method finally gets removed, it is no longer

possible to find its information on the Android developer Website [9]. Likewise, the JavaDoc for the project is not provided for removed API methods. This likely prevents the misuse of inaccessible methods, however it makes it more challenging to find migration paths for removed methods.

Our solution: Although the documentation for removed methods is not directly accessible from the Android developer site, it is accessible in the source code repositories JavaDoc. Therefore, it is possible to mine the source code history for documentation information which was removed, in order to build a complete migration picture. It could help slow adopters if Android built this information into the website as a removed section to help them migrate very old app versions.

Open Challenge #1: N-to-N API methods migrations

Description: Using the current approaches, it is difficult to assist in the migration between two sets of multiple APIs as a whole, i.e., N-to-N migration scenarios. First of all, with current techniques it is difficult to determine if a migration search is returning multiple results because of false positives or due to multiple migration paths. Secondly, understanding the relationship between the multiple APIs is challenging. Current approaches concentrate on one-to-one migration scenarios and shy away from automatically creating new source code that consists of multiple new migrated APIs. Challenging migrations are prime candidates for helping API users because of the challenges they bring. In this thesis we attempt to use existing developer knowledge to bring potential solutions to these N-to-N migration scenarios to Android API users in Chapter 4.

Table 13: Android API suggestions automatically found, compared to manually confirmed migrations.

Only methods with replacements are presented here.

API Version	Found Replacement	Missed Replacement
22	5	1
23	31	29
24	25	3
25	1	0
26	62	1

3.3.2 Step 2: Leveraging historical code-change information in API migrations

Motivation

In the previous research step, we found that a large portion of API upgrades can be recovered by searching simple keywords in documentation; only 4% of API upgrades were unrecoverable by only analyzing documentation. On the other hand, prior studies leverage software development history, such as code change per commit, when assisting in recovering API migrations [51]. For example, SemiDiff identified code changes within a commit to determine API method replacements [51].

We do not directly test any specific tool as many of them have not been maintained or require modification to run on our chosen project. Since modifying the tools could introduce errors or a bias for certain methods, we chose to test the underlying assumptions of API migration techniques in an effort to determine whether these underlying assumptions and theories hold in practice.

These techniques often assume that the removal of an existing API and the addition of an upgraded APIs exist within a short period of time (i.e. within a few commits) [51, 63, 100, 186]. Since such an assumption is heavily depended upon, yet never validated in practice, the assumption can lead to uncertainly in the usefulness of automated API upgrading techniques. Therefore, we aim to leverage historical information to recover Android API upgrades.

Approach

We first leverage code change history in the implementation of the removed or deprecated API methods. If two methods change implementation in the same commit, it is likely that their implementations are linked in some way. The more often two methods present simultaneous implementation changes, the more likely they are to share implementation details. This can allow us to determine which methods provide similar features and make links between features that would not be available by looking at release snapshots.

We collect all commits in the *git* repository of Android. For each commit, we identify the Android APIs that are changed. Since *git diff* would only provide textual based differences in a commit, we use *srcML* [46] as an intermediary to provide XML representations of Abstract Syntax Tree (AST) of the source code. By comparing *srcML* output of each source code file before and after a commit, we are able to identify

which method is changed in the commit. We then track all API implementations that are co-changed with the API implementation that is removed, modified or deprecated in the Android release. Although not all co-changes present migrations, most, if not all, migrations should present co-changes. We study whether these co-changed APIs can provide useful information for recovering API upgrades [183].

Second, for each of the known API upgrades (see Section 3.3.1), we examined the time span between its deprecation (if present), the removal of the existing API and the introduction of a new API.

Results

Over all the Android API versions studied, source code change history provides a total of 53 migration pathways. Out of these pathways, only 5 are uniquely identified by commit information. However, documentation with basic text matching identifies 119 suggestions. The Android API documentation suggestions include 90.5% of the migrations found through source code change history.

Experience #5: Historical code data, such as commits, only yields a few undocumented migration pathways and a fraction of migration pathways contained in documentation.

Existing APIs are not always deprecated, removed, or modified in the same commit as new APIs are introduced. Based on our manually identified replacements, we found that for 57.3% of them, we could not identify any commit migration pathways between the outgoing API and any replacement API.

Newly introduced APIs are often added into source code earlier than the removal or the deprecation of the existing APIs. Table 14 presents the API version difference between the appearance of a replacement method and the removal, deprecation, or change of the original method. In the studied system, 59.5% of modified methods have a replacement which appears in the same version as the modification. 10.1% of modifications have replacements outside of the Android API, and the rest of replacements are spread over the entire evolution of the API. For example, the method `getCellLocation()` was deprecated in API version 26, and was given a documented migration pathway to API method `getAllCellInfo()`. However, `getAllCellInfo()` was introduced in API version 17. Therefore, no clear migration pathway exists in API version 26 other than documentation.

In three cases, the Android API method replacement was provided in future releases. This makes it impossible to determine a replacement functionality at deprecation time for these methods, as it does not exist yet. It also makes it impossible to use commit based links since the methods clearly are modified in different releases. However, with constant monitoring of the project, it may be possible to determine a replacement through documentation and commit messages. Similarly, migration paths that appear multiple releases before deprecation time, may not be linkable through commits. Therefore, we must depend on documentation to tell us when links are created.

There are many deprecated methods left in the source code without removal. In the Android API versions studied, deprecated APIs outnumber removed APIs by a factor of 2.94. Through our research of migration methods and their emergence, we determined that there are more deprecations (244) than removed (83) and changed (142) methods in the versions studied. This presents us with an interesting finding. Only a fraction of deprecated methods was removed. This presents a contrast to Zhou and Walker [320] who show that removed API outnumber deprecated API significantly. This is not the case for the most recent Android versions. We did notice that some methods were undeprecated, such as `android.app.Notification.Builder.SetNumber(int)`, however only a few such outliers were found in the versions studied.

Experience #6: Some assumptions of history-based automated API migration are not met for the Android API, since a replacement method can be introduced earlier or later than the existing method, with a large time gap.

Challenges

Open Challenge #2: Identify the time gap between the addition and removal of APIs

Description: Our findings indicate that many API methods use migration paths that are introduced in a different version than the deprecation or removal of the targeted method. This makes it difficult to use commit based methods to identify a migration path between two methods. Table 14 shows that a large amount of modified methods have a replacement introduced earlier than their removal/deprecation. By widening the search for a migration path to a wider release cycle, it may be possible

to identify these migration paths without documentation. However, our experience shows that widening the search increases the amount of false positives. Therefore, we believe developers should minimize the use of broad time-spans when searching for method replacements and instead determine ways to optimize their historical data search through documentation informed time spans.

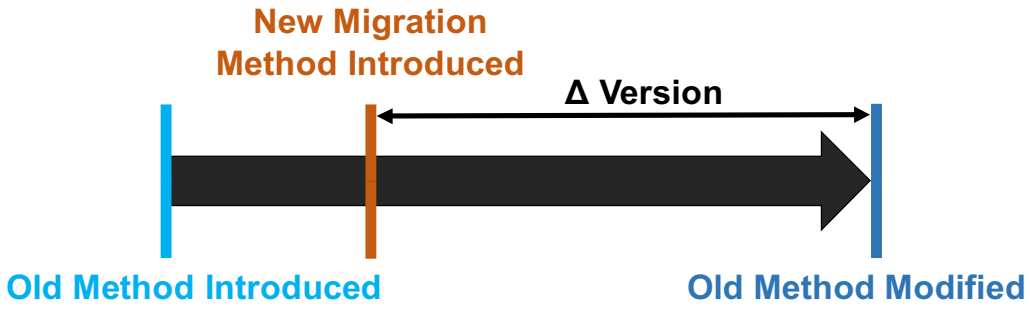


Figure 13: API migration mapping example

Table 14: API migration mapping version Δ

Negative Δ implies migration introduction before modification.
Positive Δ implies migration introduction after modification.

Δ Version	Modified Methods
-25	11
-23	1
-22	2
-18	7
-14	1
-13	3
-9	1
-8	2
-6	1
-5	8
-3	2
-2	2
-1	4
0	94
1	2
3	1

3.3.3 Step 3: API migration in FDroid apps

Motivation

In the previous research steps, we recovered API migration information using both documentation and historical code change information. However, such information may not be enough nor beneficial at all when migrating API usage in real-life Android apps. Therefore, in this step, we seek to determine how these links should be used to facilitate API usage in open-source Android apps from FDroid [79].

Approach

Android API usage in FDroid Applications

FDroid Android applications are open-source Java applications that call the Android API to interact with an Android end user. Prior studies have been done on the FDroid dataset [96]. With knowledge of the available Android API methods, it is possible to mine the FDroid applications for their API uses. We first mined the FDroid database for FDroid projects which had multiple versions and had downloadable source code. We then used a list of removed Android methods to determine the usage of removed Android methods in these FDroid projects. The list of removed methods is maintained by the Android project as part of their public framework repository [10]. Finding the usage of removed methods is done by parsing all files in all 415 FDroid projects for uses of the removed methods. We counted how many times a removed method was used, which file it was used in, and in which version of each project. We then use this information to determine the popularity of removed methods, and to find which methods are preventing an app from targeting a higher API level.

To determine whether the links produced by our approach could be useful to developers, we looked at the links between removed methods and their replacements. We gathered the uses of removed methods from a sample of 415 open source Android applications. We concentrated on Android API versions 21-25, as 66.4% of applications targeted these versions. Although Android API version 26 was used during this study, we did not have any FDroid projects with uses of the API and it is therefore not present in this research step. Since Android API version 25 only deprecated one method we do not present data related to it for clarity.

Migrating API usage in FDroid Applications

To test our suggestions in a more rigorous fashion, we use three applications which

are blocked from changing API versions due to removed methods. Using our list of FDroid application method uses, we identified three applications, *Tasks*, *Forrunners*, and *Poet-Assistant* to test our migration suggestions and attempt to migrate the applications from one version API version to the next. These applications were chosen because they presented multiple app releases (6-167), they were prevented from accessing Android API 24 due to their use of methods which were removed after API 23, and had included test suites in their development packages. We manually migrated the apps by using our suggested migration methods and ran their test suites to see if any tests were broken by our changes. We also successfully ran the apps in the Android Studio's development environment simulator as a safeguard against defective or lenient tests. We specifically attempted to target the modified functionality in the simulator, and did not experience any crashes.

Table 15: Android API methods found in FDroid projects.

API Version	Changed API Methods	Found in FDroid	Can Migrate
22	128	5	3
23	157	28	20
24	56	22	11

API levels 25 and 26 are not presented here for clarity.

Results

Only a small sample of APIs are used in FDroid projects and a small number of APIs account for the majority of the removed APIs. Not all methods which were removed by the Android development team were sampled in our study of 415 FDroid projects. Table 15 shows that between 4% and 39% of removed API methods were sampled in the FDroid projects. Therefore, not all removed methods bear the same migration weight. This implies that API architects can focus on a small amount of APIs to prioritize API migration efforts.

Experience #7 Actual modified API usage is heavily centred around a few API calls. Most API users only require support for few modified API methods.

API migrations may vary in scope. Not all migrations are equal in scope [66]. In the versions of the Android API studied we found multiple migration types.

Some migrations require the removal of one or more parameters such as `android.hardware.usb.UsbRequest.queue(Bytebuffer, int)` which became `queue(Bytebuffer)` in API version 26. The removal of a parameter could mean reworking some code if old parameters were joined.

Other migrations require the addition of one or more parameters, such as `android.text.Html.fromHtml(String)` which was changed to `android.text.fromHtml(String, int)`. The int argument was added as a way to return different flags from the `fromHtml` method. If the user wanted to keep the same functionality as the previous version of the method, the int could simply be set to a value of 0. Therefore, although inconvenient, this change is relatively simple to develop.

Similarly, some methods were simply refactored to a different class, the `FrameLayout` class `getForeground()` methods were migrated to the `View` class. For methods like these, simple refactoring could allow these methods to be migrated [66]. This is not a problem as long as the classes were not delegated outside the Android repository which happened to the `FloatMath` Android expressions which were relegated to the `java.lang.Math` package.

Comparatively, the WebViewClient method `shouldOverrideUrlLoading(WebView, String)` changed parameter type and migrated to `shouldOverrideUrlLoading (WebView, WebResourceRequest)`, and contained a very different migration strategy. The API consumer now has to learn how the new WebResourceRequest works and properly instantiate the object. This is a slightly more difficult task for a developer. A machine would almost assuredly require code examples from which to map the changes. Giving the maximum amount of available information to the API consumer in these cases allows them to determine if the migration is worthwhile. If they determine that the migration cost is acceptable, they then have access to links which could help them understand the new functionality.

Experience #8 API migrations often require further code modification than a simple rename or parameter change, e.g. object instantiation.

Developers may migrate APIs while keeping support for the old API version. In two cases, we identified applications which were using migration methods as expected and the migrations were done without any problems. However, in the case of *Poet-Assistant* we discovered that the developers were already aware of

the migration of these methods. The developers had put in conditional statements to determine the API level of the user in order to determine which API call to make. Therefore, the developers had two solutions in place for every use of the `fromHtml` method. this allowed us to determine that our suggested method was the appropriate migration method for this situation. It also presented new information that we had not anticipated. Some developers are willing to support multiple versions of the Android API simultaneously. Although *Poet-Assistant* developers had gone through the effort of making the migration to API 24, they had kept the backwards compatibility functionality for previous API levels.

Challenges

Challenge #3: Ranking migrations suggestions

Description: Using naive text matching alone, or historical source code data alone, provides multiple false positives, such as similar method names, for migration paths. Therefore, it is often challenging to rank migration suggestions in a way that makes them usable to an end user.

Our solution: By coupling both approaches we produce more accurate migration suggestion rankings. We used both approaches independently and ranked migrations based on a coupled answer from both sources of information. More accurate methods for both documentation mining and source code mining could be developed, but coupling both sources of information will lead to more accurate results than having them separate.

Open challenge #3: Identifying the existence of API replacements.

For the Android API versions studied we found that many (66.3%) API modifications did not contain migrations. It is possible that we missed migration pathways in our manual examination. However, in practice it makes little difference if there is no replacement or if the replacement is too difficult to find. Either way the API consumer does not obtain a replacement and will assume one does not exist. Therefore, we open a challenge to develop an approach to determine a gold-standard to identify the existence of migration pathways. Currently, we have no way to ascertain whether a method migration exists without documentation or architecture information, and it is possible for these sources of information to fail.

3.4 Threat to Validity

The following section aims to address the various threats to validity present in our research and how these problems were mitigated.

Construct validity.

It is possible that due to improper maintenance, documentation and source code are not representative of one another. Since all the information in this research was mined from documentation and source code history, this would cause the information in this project to be ineffective at showing the links between various methods. Any links created from unsynchronized documentation and code, would tend to arbitrary directions. We believe this to be unlikely for multiple reasons. The Android project is popular and used to support millions of apps. These millions of apps rely on and expect a high quality API as service to their businesses. It is likely that inconsistencies in documentation and source code are rapidly reported and fixed. The links produced in this study have been tested as migration links through manual inspection of both the Android framework and its documentation. No inconsistencies were noticed during this research. Our suggested method links were also tested on three different projects and the suggestions provided have been shown to compile and produce working applications.

External validity.

Since Android and Android applications were the only case studies done for this work, it is possible that the findings determined in this report are not common to other projects. Therefore, these results might not generalize to other projects. We attempt to mitigate the drawbacks of this threat by making our findings as general as possible and looking at general trends in our project sample.

We also cannot claim that the findings in this report can generalize to other programming languages. The study done in this report requires Java tooling and Java files. Therefore, it is possible that the findings in this report are only indicative of Java APIs. However, since documentation and source code repositories are not unique to the Java programming language, we believe that our findings have the potential to apply to APIs from any programming language.

Internal validity.

The findings in this report are all based on documentation that was accepted by the Android development team. This might present a bias, as it is possible that the

Android development team uploads only documentation that is favourable to them and removes older documentation to hide any mistakes or risks caused by erroneous documentation. We did not observe documentation maintained outside of the source code repository. This was mitigated by looking at all available commits, which should present all changes in documentation. If any inconsistencies in documentation exist, they should appear in the committed changes to the documentation, and none were noticed. If the Android development team only presents filtered commits which have perfect documentation, we must accept the available information at face value as we have no other sources of information about the API.

3.5 Chapter Summary

In this chapter, we presented our experience in using Android API source-code repository coupled with documentation to provide suggestions for Android API migration. During our practice, we find that although a portion of the removed or deprecated API methods do not have a replacement, and that identifying a replacement using documentation or historical code change information is not a challenging task for Android API users. In particular, Android official online documentation provides valuable information and enables the use of simple keyword searches to find a replacement for removed or deprecated API methods. Existing tools such as ARENA [227] could mine this information. However, when we applied API method replacements to migrate Android API methods in FDroidApps, we experienced other challenges, which are more time consuming to address, such as initializing new parameter types. We document these challenges so that future research can investigate them and propose automated techniques to assist in API migration. This thesis attempts to propose a technique to remedy some of these challenges. The proposed technique is presented in Chapter 4 of this thesis.

This chapter highlights some failings of current API migration techniques and provides opportunities to improve the understanding of API evolution, particularly API migration. Several challenges to be tackled by future research have been presented based on our experience with the Android API and FDroid apps.

Chapter 4

Using Existing API User Knowledge as Android API Migration Aid

The fast-paced evolution of Android APIs has posed a challenging task for Android app developers. To leverage Androids frequently released APIs, developers must spend considerable effort on API migrations. Prior research and Android official documentation typically provide enough information to guide developers in identifying the API method calls that must be migrated and the corresponding API calls in an updated version of Android (*what* to migrate). But developers lack the knowledge of *how* to migrate the API method calls. There exist code examples, such as Google Samples, that illustrate the usage of APIs. We posit that by analyzing the changes of API method call usage in Android code examples, we can learn API migration patterns to assist developers with API method call migrations.

In this chapter, we propose an approach that learns Android API migration patterns from code examples, applies these patterns to the source code of Android apps for API method call migration, and presents the results to users as potential API method call migration solutions. To evaluate our approach, we migrate API method calls in open-source Android apps by learning API migration patterns from code examples. We find that our approach can successfully learn Android API migration patterns and provide Android API migration assistance in 71 out of 80 cases where Android apps require API method call migration. Our approach can either migrate

Android API method calls with little to no extra modifications needed or provide guidance to assist with the migrations. Through a user study, we find that adopting our approach can reduce the time spent on migrating Android APIs, on average, by 29%. Moreover, our interviews with app developers highlight the benefits of our approach when migrating APIs. Our approach demonstrates the value of leveraging the knowledge contained in software repositories to facilitate API migrations.

4.1 Introduction

Existing migration recommendation techniques [52, 100, 186, 296] typically focus on identifying *what* is the replacement of a deprecated API (e.g., one should now be using `methodB` instead of `methodA`), instead of *how* to migrate the API calls for the replacement (e.g., how to change the existing code to call `methodB`). However, a recent experience report shows that all too often, Android API official documentation clearly states *what* to replace for a deprecated API, while actually performing API migrations is still challenging and error prone [135].

There exist many publicly-available code examples online illustrating API usages. As an example, Google provides a set of sample Android projects on the Google Samples repository [93]. Developers often study these sample projects and other code examples (e.g., code from open-source Android apps) to help them with API migration [150, 268, 279, 281]. Nevertheless, studying the code examples to know the changes needed for API migrations is a manual and time-consuming process. Furthermore, identifying *where* and *how* to apply migration changes puts an extra burden on developers during software maintenance.

```
  view.setFocusableInTouchMode(true);
- view.setBackgroundColor(mFrag.getresources().getColor(R.color.default_background));
+ view.setBackgroundColor(mFrag.getresources().getColor(R.color.default_background, null));
  view.setTextColor(Color.WHITE);
```

Figure 14: An API migration example of the *getColor* API, in the *GridItemPresenter* class of *androidtv-Leanback* project in commit *6a96ad5*.

In this chapter, we propose an approach, named A3, that mines and leverages source code examples to assist developers with API migration. We focus on Android API method call migrations, due to Android's wide adoption and fast evolution [162]. Our approach automatically learns the API migration patterns (i.e., pairs of ASTs

before and after an API method call migration of the same piece of code) from code examples taken from available code repositories, thereby providing varied example patterns. Afterwards, our approach matches the learned API migration patterns to the source code of the Android apps to identify suitable learned API call migration candidates. If migration candidates are identified, we apply the learned migration patterns to the source code of Android apps, and provide the resulting API method call migration to developers as a potential migration solution.

To evaluate our approach, first A3 learns Android API method call migration patterns from three sources of code examples: 1) official Android code examples provided by Google Samples [93], 2) migration patterns that are learned from the development history of open-source Android projects i.e., FDroid [79] and 3) API migration examples that are manually produced by a sample of Android API users. Our approach then applies Android API migration patterns to open-source Android apps from FDroid and we leverage their test suites and manually run the apps to validate the correctness of the migration. Furthermore, we compared our approach to LASE [165], a tool meant to apply code edits learned from examples. Moreover, we carry out a user study to determine the actual and perceived usefulness of our approach. In particular, we answer three research questions.

RQ1 *Can we identify API migration patterns from public code examples?*

Our approach can automatically identify 80 migration patterns with 96.7% precision in Android APIs used in public code examples, and obtains a recall of 97% using our seeded repository.

RQ2 *To what extent can our approach provide assistance when migrating APIs?*

Based on 80 migrations candidates in 32 open-source apps, our approach can generate 14 faultless migrations, 21 migrations with minor code changes to the API user application, and 36 migrations with useful guidance to developers. Furthermore, interviews with four developers highlight a positive developer response to our Android API method call migration patterns.

RQ3 *How much time can our approach save when migrating APIs?*

Through a user study with 15 participants and 6 API migration examples, we show that our approach provides, on average, a 29% migration time improvement and is seen as useful by developers.

Previous research has proposed approaches such as Sydit [164] and LASE [165] to help developers with API migration; however, these approaches must be manually pointed towards pre-migration examples without the ability to automatically retrieve or identify them [232]. Furthermore, the effectiveness of code examples on migration is affected by the context of the examples, whereby examples with closer contexts will waste less developer time by reducing time spent testing barely related cases [232]. Therefore, by considering multiple examples from different contexts our approach generates well-fitted migration solutions.

Our approach can be adopted by Android app developers to reduce their API migration efforts to cope with the fast evolution of Android APIs. Our approach also exposes the value of learning the knowledge that resides in rich code examples to assist in the various tasks of API related software maintenance.

Chapter Organization. Section 4.2 provides a real-life example of an API migration to motivate this study. Section 4.3 describes our automated approach, A3, that assists in Android API migration. Section 4.4 presents the design of the experiments used to evaluate our approach and Section 4.5 presents the results of our experiments. Section 4.6 describes threats to the validity of this study. Finally, Section 4.7 concludes the chapter.

4.2 A Motivating Example

In this section, we present an example, which motivates our approach based on learning migration patterns from code examples to assist in API migration.

In Android API version 23, the *Resources.getColor* API method (as shown in Listing 4.1) was deprecated and replaced (as shown in Listing 4.2). In fact, the deprecation and replacement (*what* to migrate) are clearly shown in the official Android documentation [8]. However, even with the help from the documentation, the addition of a new parameter provides new challenges to the Android app developers. Because they may not have the knowledge necessary to retrieve the *Theme* information nor to initialize a new object for the *Theme* to make the new API call. Moreover, since the old API call does not require any *Theme*, even if developers can provide a *Theme*, there is no information on how to preserve backward compatibility.

On the other hand, there exists open source example projects on Google Samples [93], i.e., *androidtv-Leanback*, a project on Github that presents several uses of the *Resources.getColor* method. With the introduction of a new Android API version, these code examples were also updated. By looking at an example, we find that it clearly demonstrates how to call the *Resources.getColor* API method (see Figure 14). From the changes to the code example, we can see that to maintain backward compatibility, developers can simply pass a *null* value to the API. Without such an example, figuring out that a *null* value preserves backward compatibility would require trial and error from developers. By learning such a migration pattern in the code examples, the effort of the challenging task of *how* to migrate an API method call can be reduced for developers.

Listing 4.1: Resources.getColor API method before migration

```
public class Resources extends Object {
    public int getColor(int id)
}
```

Listing 4.2: Resources.getColor API method after migration

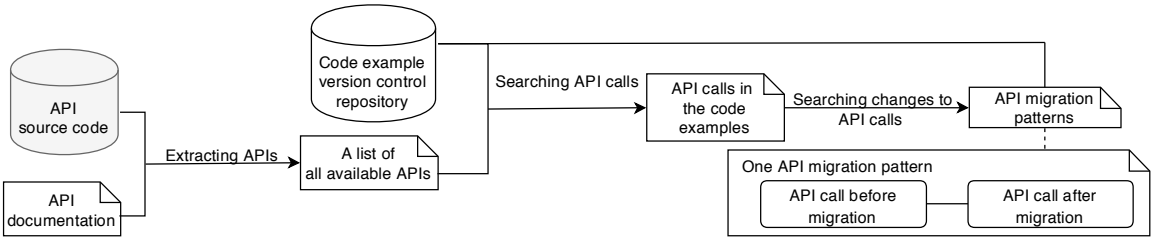
```
public class Resources extends Object {
    public int getColor(int id, Resources.Theme theme)
}
```

However, finding these migrations on Google Samples is a laborious endeavor. First of all, the code examples do not index their usage of APIs. Developers may must search for the API usage of interest from a large amount of source code. Second, even with the API usage in the code examples, developers need to go through the code history to understand the API migration pattern, i.e., *how* to apply these changes, which can be much more complex than the aforementioned example. Finally, even with the migration pattern, developers need to learn how to apply the migration on their own source code, which can be a challenging task [135].

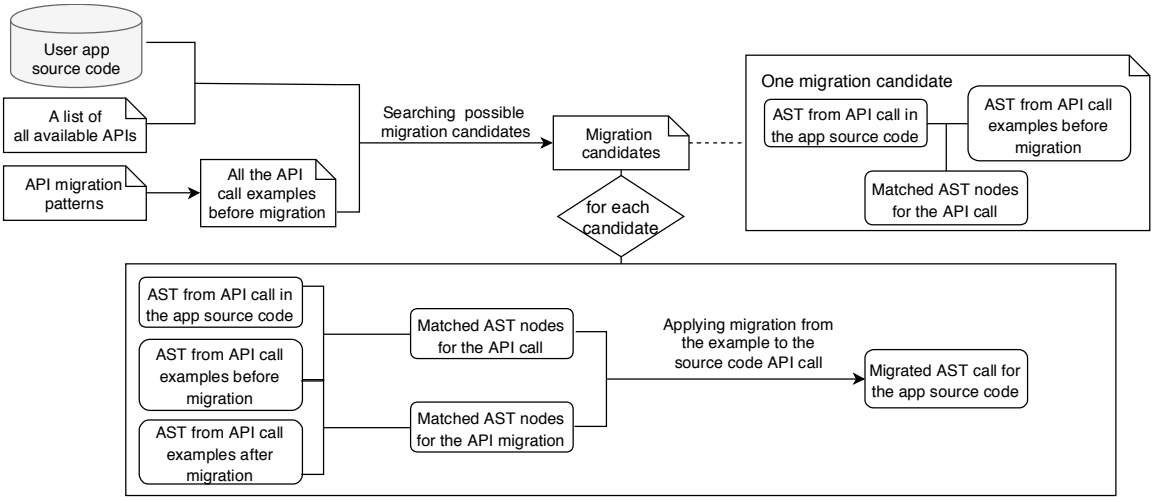
Taking the aforementioned API method *Resources.getColor* as an example, we can detect a total of 1,626 places where the *Resources.getColor* Android API method is called in its deprecated form on a sample of 1,860 open source Android apps from FDroid [79]. Migrating *Resources.getColor* to Android API version 23 or later for all of those apps requires a significant amount of effort. Automating the above-described

migration approach and providing the information of the learned pattern to developers can significantly reduce the required migration effort.

4.3 Approach



(a) Step 1: Learning API migration patterns from code examples



(b) Step 2: Applying learned API migration patterns to API calls in the source code

Figure 15: An overview of our approach, named A3, that assists in API migration

Our approach (A3) consists of two steps: 1) identifying API migration patterns from code examples and 2) applying the migration pattern that is learned from the example to the source code of the Android apps. An overview of our approach is shown in Figure 15 and an implementation of our approach is publicly available online[1].

[1]The repository which contains the tool and data presented in this paper can be found at: `https://github.com/LamotheMax/A3`.

4.3.1 Learning API migration patterns from code examples

In the first step of our approach, we mine readily available code examples to learn API migration patterns. Such code examples can be found through online code repositories such as GitHub and FDroid [79]. Our approach also supports self-made examples, which allows users to produce their own migrated stubs, or use their own projects as data to feed forward into other projects.

Extracting APIs

We automatically extract all the Android APIs for every Android version that is available from the official Android API documentation. However, the Android documentation may have discrepancies to the APIs in their source code[2]. For every version of Android, we obtain the Android source code and parse the source code using Eclipse JDT AST parser [25]. We identify a list of all public methods as available APIs.

Searching API calls from code examples

```
Input  : Code example repository exRepo and available API list apiList
Output: Migration example
/* First do basic lexical matching */
foreach api in apiList do
    apiCallData ← api.get(callData)
    apiParams ← apiCallData.getParamCount()
    apiName ← apiCallData.getName()
    potentialMigrations ← null
    foreach commit in exRepo do
        potentialMigrations ← commit.getAPIcalls(apiName, apiParams)
    end
    /* Selective check for all potential migrations */
    foreach call in potentialMigrations do
        callAST ← call.buildAST()
        if callAST.matches(apiCallData) then
            saveExample(call)
    end
end
```

Algorithm 1: Our algorithm for searching API migration examples.

[2]We reported two discrepancies between the Android API documentation and the code repository.

In this step, we identify the API calls that are available in the code examples. A pseudocode algorithm of this step is presented in Algorithm 1. We may design an approach that builds AST for all available code examples as well as the targeted Android app source code. Afterwards, we could use the ASTs to match the API calls in the source code to the code examples. However, such an approach would be time consuming due to the fact that 1) building ASTs is a time-consuming endeavor, 2) the complexity of AST matching is non-negligible and 3) the number of Android APIs that are available is large.

To reduce the time needed to identify API calls, and thereby reduce the challenges faced by developers who seek to migrate APIs, we first use basic lexical matching to limit the scope of needed AST building and matching. We first search all the files in the code example for strings that match API names. This technique allows us to quickly get a preliminary list of potential API calls. These files are then selected for further processing as potential matches. Although the basic lexical matching can lead to false positives, the goal is reducing the search space and the following AST matching can remove these false positives. For example, in Figure 14 we specifically search for the keyword *getColor*, although it is possible for this basic search to falsely identify a similar post-migration API as a migration target, once the refined check with AST matching is complete, i.e., these *getColor* false positives would be filtered out if they did not have exactly one integer parameter, the right return type, and all other discernible AST properties for *getColor*.

After the basic lexical matching, we apply AST matching on the files with potential matches. This allows us to scope down the previously identified results to obtain high precision findings without sacrificing too much performance. We leverage JDT [25] to parse the source code in the code examples to generate their corresponding ASTs. For each method invocation in the AST, we compare with the APIs that are potentially matched from the lexical search. If the AST can be fully built, we aim to obtain the perfect matches between method invocation and API declaration. In particular, a perfect match requires matching the method invocation name to an API declaration as well as having perfectly matched types for all parameters. In some cases, the code examples may not be fully complete (e.g., missing some external source code files or dependencies), leading to partially built ASTs. With the partially built file level ASTs, we consider a match if there exist the correct import statement of the

API, the correct method name, and the correct number of parameters. All method instances are saved, along with their invocation string. This data can quickly be reviewed by a developer to determine if a false positive was detected, making our approach transparent and understandable. Developers interviewed in RQ2 presented in Section 4.5, were presented and reviewed the migration instances that were related to their familiar app and confirmed our results.

Learning API migration patterns

In this step, we search API calls for every version of the code examples. For instance, if code examples such as Google Samples [93], or FDroid projects [79] are hosted in version control repositories, we detect API calls in every commit of the repository.

Searching every commit for a multitude of projects requires surmounting the challenge imposed by the large scale of available data from API user projects. We surmount the challenge imposed by the large amount of data available by leveraging the basic *diff* in the version control system (like Git) to determine in which commits a specific string is modified to collect commits that contain changes to API calls.

Commit level migrations allow us to reduce the amount of code modifications that could obfuscate an API migration. If we reduce the search scope to a granularity coarser than commit level we might miss certain migrations. Similar to the lexical search from the last step, the use of a diff tool is merely scoping down our search space. For the commits that potentially impact API calls, we parse the AST of the changed files in the commit. We compare the ASTs generated from the source code before and after the commit. If the API invocations in the AST are modified in the commit, we consider the commit as a potential API migration pattern. Hence, each API migration pattern consists of an AST built from the example before the migration and an AST built after the migration.

For the code examples that are hosted as text files outside of repositories, we apply a text diff on each two consecutive versions of the example instead of using commits to scope down the search space. The secondary AST matching step on these text files is identical to that from the version control repositories.

4.3.2 Applying learned API migration patterns to API calls in the source code

In our second step, we collect all the API migration patterns from the first step and try to apply these patterns to the API calls in the Android app source code.

Searching possible migration candidates

Similar to our first step, to resolve the challenges imposed by the scale of the problem at hand, we reduce the search space, our approach first uses API names to lexically search for API calls with available migration patterns in the source code of the targeted Android app.

Migrations can be dependent on the context of surrounding code, we cannot assume that one migration example will suit all possible use cases. For example, API such as *Resouces.getColor(int)* can be present in a variety of use cases. To match valid migration patterns, we must not only match API calls, but also determine if an example matches a user's usage of the API before migration, in order to determine if we can migrate it. We leverage data-flow graphs to match the API calls in the migration patterns and the targeted Android app sources. We construct a data-flow graph from the API call example "before" the migration in the migration patterns. We also construct data-flow graphs in the API calls in the Android app source code. Only if the data-flow graph from the example is a subgraph of a potential API call in the Android app source code do we then consider this a migration candidate. This allows us to assume that the example API being used as a migration pattern has a similar use case to the API call in the targeted Android app.

Applying the migration from the example to API calls in the source code

Existing API migrations can contain implementation details that cause incompatibilities with other projects. Therefore, our approach must mitigate the challenges imposed by implementation details and allow developer interaction. We employ data-flow graphs, rely on the large number of migration examples provided by open-source projects to obtain our migration examples, and let the developer have final say at every step of the approach.

If any migration candidate is found, we then attempt the migration. We first

```
Input  : Migration mapping mappedDFG and client data-flow graphs
         clientDFG
Output: Migrated data-flow graph
/* Traverse all data − flow graphs */
foreach DFG in clientDFG do
    DFGMap ← mappedDFG.get(DFG)
    changedAPIs ← DFGMap.getChangedAPI()
    /* Migrate all migrateable APIs */
    foreach changedAPI in changedAPIs do
        changedNodes ← DFGMap.getDataLinks(changedAPI)
        missingNodes ← DFGMap.getNodesToAdd(changedAPI)
        /* Adjust the data − flow graph */
        DFG.addNewNodes(missingNodes)
        DFG.migrateDFG(migrateableNodes)
    end
end
```

Algorithm 2: Our algorithm for migrating code.

compute the migration mappings by comparing the examples before and after API migration. This mapping contains any changes that must be made to existing code statements, obtained by comparing the names and types of each code statement in the data-flow graph. The migration mapping also contains any new code statements that are present in the "after" migration example but were missing in the "before" migration example.

In order to obtain an accurate migration mapping between the "before" and "after" examples (i.e., changes that were made to migrate the API), we need to eliminate the changes on AST that are not related to the API migration. We achieve this by relying on the data-flow graphs that are built from the examples. We first remove all the nodes in the data-flow graph where all the associated nodes are perfectly matched between the "before" and "after" examples. Because they are perfect copies of one another, those nodes cannot contribute to a migration. We keep the nodes in the data-flow graphs that remain unmatched and are associated with the node that is of interest to the API call. Finally, we compare the nodes that are kept to find the matched data-flow graph for the API call in the Android app after migration.

Once we obtain the most likely migrated data-flow graph in the "after" API migration example, we produce a backward slice of the data-flow graph starting from the API call. In other words, we only look at nodes that give data to the API call.

Based on our sliced data-flow graph, we then map each node in the "before" example to the most closely matched node in the "after" example. Any unmatched data linked nodes are considered to be new nodes and are saved to be added during migration.

Finally, we use the migration mapping to transform the project source code into the "after" API migration example. Pseudocode of our migration algorithm can be found in Algorithm 2. The transformation also looks for any object types that are matches between the "before" example and the project source code to infer the names used in the Android app source code, to prevents the introduction of new variable names. Our approach can produce migrations that are interprocedural and intraprocedural. However, we limit the migration scope of our approach to single files because prior work has found that field and method changes account for more than 80% of migration cases [48]. Adding migration across multiple files would increase the chance of making mistakes, both when mining and applying migration patterns, and only account for a minority of migration cases.

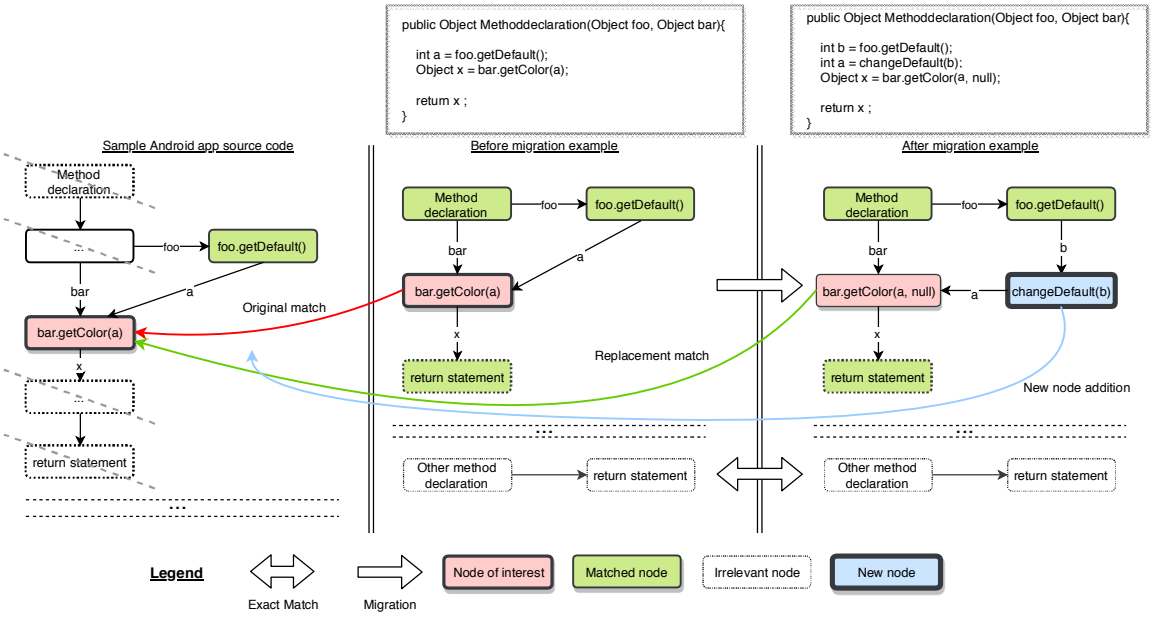


Figure 16: Example of applying a migration from an example to Android app source code

To better explain our approach, Figure 16 illustrates an example of applying a migration from an example to an API call. The *before* and *after* migration examples illustrate potential example code obtained from sample projects. We can see that methods that are exact matches are seen as irrelevant because they add no information

to the migration. Similarly, we can see that nodes after the migration node of interest, in this case a return statement, are also seen as irrelevant nodes. This is because it does not matter what the user does with their code after the API call (i.e., not related to the usage of the API). Other nodes are then matched between *before* and *after* examples to determine which node is most likely the migration. Any nodes that do not obtain a match are considered new nodes required for migration and will be added in user projects, as denoted by the color blue in our example. Once a match is produced between migration examples, the migration mapping can be applied to the user example, as presented by the arrows in Figure 16 and as explained by the pseudocode in Algorithm 2.

4.4 Evaluation

To determine the extent to which our approach, A3, can be used to assist with API migration, we conducted a series of experiments. We first present the research questions to evaluate our approach. Then, we present our data acquisition approach for Android apps and code examples.

4.4.1 Research questions

We evaluate our approach by answering three research questions. We present our research questions and their motivations in the rest of this subsection.

RQ1 *Can we identify API migration patterns from public code examples?*

Our approach uses code examples to automatically learn API migration patterns. Therefore, the availability of migration code examples is very important to our approach. In this RQ, we study how many API migration patterns we can find in publicly available code examples and calculate the *precision* of our approach. We also enlisted the help of 20 volunteer developers to produce a sample migration dataset in order to obtain a known number of migration examples and determine the *recall* of our approach.

RQ2 *To what extent can our approach provide assistance when migrating APIs?*

In this RQ, we use the publicly available and user produced examples to automatically learn and apply migration patterns. We evaluate and quantify the extent to which our approach, A3, can leverage these examples to assist developers in migrating

API calls, though comparisons with an existing approach and through testing and manual evaluation of Android app migrations.

RQ3 *RQ3: How much time can our approach save when migrating APIs?*

In this RQ, we enlisted 15 developers to participate in a user study to evaluate the time savings provided by A3. We timed the participants while they migrated an amalgamation of six API examples, for which they randomly received help from A3. The participants were also asked to rate the usefulness of our approach.

4.4.2 Data acquisition

We present the Android apps that we want to migrate and the sources of our code examples.

Android apps for migration

We selected our Android apps through the free and open source repository of Android apps: FDroid [79]. We clone all the git repositories of FDroid apps that are hosted on GitHub and implemented in Java. In total, we obtained 1,860 apps. From these apps, we chose the ones that are still actively under development, since they are more likely to benefit from migrating to the most recent Android APIs. Therefore, we only selected apps that had code committed in the six months prior to our study. In order to assist in later verifying the success of the migration, we selected only apps that contained available tests. Because we could not ascertain that the tests would guarantee that the migrated API code would be tested, we only selected the apps that could be built with the official Android build system, Gradle, so that we could verify the functionality of apps after migration. We focused the evaluation of our approach for API migration on the 10 latest versions of the Android API because they account for 95.6% of Android devices in the world [7]. Therefore, we selected the apps that target the 10 latest versions (API versions 19-28) of the Android API. This allowed us to collect a sample of 164 FDroid apps on which to test our migrations. All the other apps (1,696) were used to extract API migration examples.

Sources of public code examples

Our approach relies on code examples to learn API migration patterns. In the evaluation of our approach, we focus on two sources of public code examples: 1) official Android examples, i.e., Google Samples [93] and 2) FDroid app development history. We also use Android API usage patterns extracted from our FDroid sample to construct the original examples given to our user study participants.

Google Samples. The Android development team provides code samples to assist app developers understand various use cases of the Android APIs. This code sample repository, aptly named Google Samples, contains 234 sample projects, 181 of which are classified as Java projects [93]. We mined the Google Samples repository, hosted on GitHub, for Android API migration examples.

FDroid app development history. Due to the widely available open source projects, if one open source project migrates a deprecated API in their source code, other developers may leverage that migration as an example. With the publicly available development history of FDroid, we can leverage the API migrations that exists in the FDroid apps as code examples.

Source of private code examples

It is impossible to determine how many migrations have been produced and can therefore be recovered from mining public repositories, therefore we cannot use this data to test the recall of our approach. We therefore produced a sample migration dataset with the help of volunteer developers. This dataset was used in RQ1 to determine the recall of our approach when mining for migration examples.

We developed a dataset with the help of 20 participants (14 graduate students and six professional developers) to create migration examples for Android API methods that had known migrations. Based on our study of the FDroid repositories, we selected the 30 Android APIs most frequently called but without public examples of migration in FDroid apps. We manually created stub classes based on multiple examples of existing code taken from FDroid apps to reduce the amount of code our volunteers had to read. We then presented these classes to our participants as files sampled from real apps that needed migration. The participants were each given five files, selected at random from our pool of 30 Android APIs that needed migration, in order to guarantee multiple samples of successful migrations for each API. Therefore,

each API had at least three participants attempt the migration. The participants were told which API to migrate for each file, and were allowed to use the official documentation or any other source of information they deemed necessary to create suitable migrations. We manually verified the results of the participants and randomly selected working examples to seed our sample repository of 30 real migration code examples that were created by real developers and sampled from real projects[3].

4.5 Evaluation Results

In this section, we present the results of our evaluation by answering three research questions.

RQ1: Can we identify API migration patterns from public code examples?

We focus on two sources of examples in this RQ: 1) official Android examples, i.e., Google Samples [93] and 2) FDroid app development history. We measure the prevalence of API migration examples by determining the number of API migration examples that we can mine from our example sources. We only consider the APIs with migrations that have officially been documented by Android developers to validate our results.

In the 10 latest versions of Android (versions 19-28), we were able to manually identify 262 APIs which had documented migrations. Out of all the Android APIs with migrations, only 125 of those occurred in our sources of examples.

Table 16: API migrations identified

Data sources	API identified
FDroid API migrations	79
Google Samples API migrations	10
Total distinct API migrations found	80
Total API uses found in apps	125
Total possible	262

By searching for API migrations (c.f. Step 1 in our approach in Section 4.3.1) in the Google Samples [93] and the development history of FDroid apps, our approach

[3]The 30 code examples, and their migrations, are available as part of our replication package

can automatically identify 10 and 79 API migration examples out of 125 API occurrences, respectively. Among those migration examples from Google Samples, only one API is not covered in the FDroid examples, giving us a total of 80 distinct API migrations found (see Table 16). However, we note that the nine other examples from Google Samples that overlap with other sources are still useful due to the possible minor variations in ASTs that can occur when migrating. Some examples may provide a more suitable migration with less needed human effort from developers (c.f. RQ2 & RQ3). We manually analyzed all examples mined and determined that our approach was able to find migration patterns with 100% precision in Google Samples (i.e., 10/10 examples correctly identified), and 96.3% precision in FDroid apps (i.e., 79/82 examples correctly identified), giving a total precision of 96.7%. Some examples are migration false positives due to user modifications that can be mistaken an migrations, we discuss examples that are not migrations in detail in RQ2.

To calculate the recall of our approach, we used the manually created sample of 30 Android migration examples that was created by volunteer developers (c.f. Section 4.4.2). Our approach was able to successfully extract 29 out of 30 migration examples from the seeded repository, giving us a recall of 97%. The example which could not be extracted was due to line-breaks within the API call. It would be possible to fix this corner case either by preprocessing all commits to remove line-breaks, or modify our API search to allow for arbitrary line-breaks.

Table 17: Classified API migration patterns identified by our approach

Migration difficulty	Type	Frequency
EASY	Encapsulate	6
	Move method	4
	Remove parameter	1
	Rename	7
MEDIUM	Consolidate	7
	Expose implementation	10
HARD	Add contextual data	23
	Change type	15
	Replaced by external API	7

We manually classified the 80 distinct migration patterns identified by our approach according to pre-established API migration classifications [48]. Table 17 presents our classified patterns. As can be seen in Table 17, the majority (45/80)

of the migrations we identified are considered hard to automate [48]. This implies that our approach can surmount hard technical challenges, as well as find easy-to-migrate examples. The easy difficulty migration implies that the migration patterns can normally be completed by most IDEs such as Eclipse [48]. For example, in API version 26, *Notification.Builder.setPriority* was moved and renamed to *NotificationChannel.setImportance* and *CookieSyncManager.startSync* is no longer useful since it was encapsulated by the WebView class in API version 21 and is now automatically done by that class. Medium difficulty migrations (i.e., migrations previously considered partially automatable) were also found. We identified examples of API consolidation such as *NetworkInfo.getTypeName* which was consolidated into the *NetworkCapabilities* class in API version 28 and must now instantiate that class and call a different method *hasTransport* new with constants. We also identified cases where implementation was exposed to give more control to the API users. For example the *Notification.Builder.setDefaults* method started allowing users to enable vibration, enable lights and set sound separately in API version 26 rather than be done as part of one method, as it was done before API version 26. This means that our approach must be able to find code to instantiate all of these new functionalities. As previously mentioned, our approach also identifies hard to automate migration patterns. We identify patterns such as added contextual data in methods like *Hmtl.toHtml* where new parameters were added in API version 24 to allow for more options and user control. In most of these cases a default value for new parameters is allowed, which our approach can mine from existing projects to allow users to migrate their project. Changed types such as faced when migrating to *Message.getSenderPerson* from *Message.getSender*, in API version 28, can be handled by our approach by following the control flow graph of calls and modifying the calls appropriately based on previous migrations. Finally, APIs replaced by external APIs such as the FloatMath APIs in API version 22 *cos, sin, sqrt* can be changed to their new library format, and the import statement can be created. The user then only has to make sure that the new API library is included in their project.

Our approach can automatically identify 80 migration patterns with 96.7% precision in Android APIs used in public code examples, and obtains a recall of 97% using our seeded repository.

RQ2: To what extent can our approach provide assistance when migrating

APIs?

In this RQ, we examine whether our approach can provide assistance to API migration in Android apps. In particular, we apply our approach to migrate 80 API calls in 32 FDroid apps (based on both the migration patterns learned from public code examples and user generated examples). In order to examine whether a migration is successful, we leverage the tests that are already available in the apps and collect the ones that can exercise the migrated API calls. In order to avoid tests that are already failing before the migration, we run all the tests before the migration of the apps and only keep apps with passing builds. The apps that we selected had an average of 10 tests, with a minimum of 6 tests and a maximum of 52. Afterwards, we try to build the migrated apps. For the apps that can be successfully built, we run the collected tests again to check whether the migration is completed successfully. Furthermore, we also manually install and run the migrated apps to ascertain that the migration does not cause the app to crash. Furthermore, we provide comparisons with prior approaches such as LASE [165].

Previous studies have shown that producing exact automated migrations is a difficult task [48]. We consider this fact and attempt to mitigate it through the use of our approach. However, in cases where it is possible to present an exact migration to a user, such a task should be attempted to save development time. Therefore, in the best case scenario we attempt to provide an exact and automatic migration to users. Table 18 summarizes the results of our migrations. Our approach can provide assistance in 71 out of 80 migrations through faultless migrations, migrations with minor modifications, and through the examples we suggest when we experience unmatched guidance or unsupported cases. The following paragraphs discuss each different type of result in detail.

Faultless migration. If all the tests are passed, and the app runs, without any further code modification, we consider the migration as exact. We succeeded at giving users an exact migration in 14 cases. Nine of these migrations were learned from the manually produced code examples from the examples manually created by participants. However, we were able to use five examples from FDroid app development history to produce faultless migrations. Such a result tells us that given a rich example, it is possible to provide fully automated working migrations.

Migration with minor modifications. In cases where an app, after the migration,

Table 18: Automated migration results based on migration patterns learned from three different sources

	FDroid	Google Samples	Volunteer developer sample
Faultless migration	5	0	9
Migrated with minor mod.	5	4	12
Unmatched guidance	18	1	9
False positive	0	0	3
Ex. was not a migration	6	0	0
Unsupported cases	7	0	1
Total migrations	*41*	*5*	*34*
Distinct API	*21*	*4*	*15*

does not build and run, or does not pass the tests, we manually checked the error message. In such cases, the migration pattern may be correct, yet our automatically generated migration may need minor modifications to build and run the app and pass the tests. For each case, we determined a way for the migration to succeed with minimal code modification. An example of such modification can be adjusting a variable name. We were able to provide 21 migrations with minimal modifications. We consider the number of tokens that must be changed for the migration to be successful. The number of tokens changed was determined by post-modifications performed by the first author, who manually went through failed automated migration cases, modified the app's code to make the migration successful and measured the size of the modification. We use the absolute number of tokens changed rather than a percentage of tokens matched since the automatically matched tokens do not require any effort. We consider any modifications to a code token as a token modification.

We found that the modifications needed for an imperfect but successful migration requires modifying between one to seven tokens (3.65 tokens on average). We consider the amount of effort needed on such modifications rather small, especially since they are mostly simple renames and the addition or removal of keywords. The brunt of the work, namely finding a migration candidate, finding a migration pattern, and matching this pattern, is provided by our approach, A3. Therefore, the user can simply "glue" any unattached pieces of example code into their application.

We examined the different scenarios that require minor modification to qualitatively understand the effort needed for such modifications. In total, we identify four

reasons for such modifications: *variable renaming*, *missing the keyword* ***this***, *wrongly erasing casting*, and *removing API calls*.

As a design choice, we opt to conservatively not remove any API calls for migration, since mistakenly removing API calls may cause large negative impact to developers. Instead, our approach tells the users that a change must be made to the API, and the API call must then be manually removed. Although this may require the manual modification of removing several tokens, the effort of the change is minimal. We experienced three *removing API call* cases. For example the *View.setDrawingCacheQuality(int)* method was made obsolete in API 28 due to hardware-accelerated rendering [9]. This means that old code referring to drawing cache quality can be simply deleted, as the OS now handles this through hardware.

Unmatched guidance. It is possible for our approach to fail to match an example to a known migration (see Section 4.3.2-1). These cases exist due to the nature of our example matching. Since we consider the data-flow surrounding a migration, our examples must contain a similar data-flow graph to the Android app considered for migration. We conservatively opt for such an approach to reduce the number of falsely generated migrations, since they may introduce more harm to developers than assistance. As a result, if the Android app instantiates their API call in a different way than our examples, our approach will not attempt to automate the migration. For example, if the user uses nested method calls inside an API call and we cannot reliably map the return types of the nested method calls, we will not consider the code to match. Examples that contain unmatched but similar API migrations will be presented to the user, and they can choose the correct migration and manually apply it afterwards.

For the 28 migrations for which none of our examples contained a match, 18 of these are from FDroid development history. This implies that the FDroid API usage is more often tailored for the apps' needs and is not often coded in a general manner. Future research may investigate automatically generalizing the usage of API to address this issue.

False positives. We consider a migration to be a false positive when our approach presents an unnecessary migration. This would occur if a migration example were erroneously matched to a non-migrateable method invocation in Android app code. This only occurred three times in our tests and the occurrences were all from the

manually produced examples. We believe this to be the case due to the simplicity of the manually produced examples. Since the examples are simplified and contain very little context code, the corresponding data-flow graphs are often simple. This allows the examples to be used in a wider range of situations than the mined code samples. However, this leads to false positives as a trade-off, especially if method names are commonly used and few parameters are used (e.g. *setContent()*). These false positives can be caught at compile time since they would not compile, and can therefore be corrected or discarded by the developer without harm to the app.

Example was not a migration. As previously mentioned, since we obtain our migration examples through automated tool assistance (c.f. RQ1), it is possible for our approach to present faulty migration examples. This normally occurs when a deprecated API call is modified, but not migrated to the updated version of the API. An example of such a modification would be to rename the variables in the API call. This change would be mistakenly identified by our approach as a modification to an API call. Therefore, if such an example were to be used to attempt a migration, we would present a migration with no effect, and therefore cause no harm to the Android app. We experienced six such instances in the FDroid examples. However, we did not encounter these in our Google Samples nor the manually produced examples.

Unsupported. There exist a few unsupported corner cases that our approach would not support. For example, we cannot automatically produce migrations if the API call is spread across a try catch block, or within a loop declaration or conditional statement declaration. As with unmatched guidance cases, we provide the user with migration examples that may be relevant to their migration, instead of attempting to provide automated migration on their code, therefore no harm is done to the code. We experienced eight such cases.

Table 19: Comparison with LASE [165]

	A3	**LASE**
Total Migrations Possible	**17**	
Migration map successfully created	17	15
Migration point successfully found	9	1
Migration faultlessly applied	7	1

Comparison with other approaches. To the best of our knowledge, there currently exists no other approach that can automatically identify API migration examples and automatically suggest and apply them to app code. Tools such as SEMDIFF [52], and AURA [296] find migration locations, but do not provide example-based automatic migrations. We find 97% of migration locations based on our examples, which is close to the upper bound of what these approaches can achieve. We selected LASE [165] as a more direct comparison for our approach since LASE resembles the automatic migration of our approach while needing to be provided compilable examples. Due to LASE's need for compilable examples, we had to use A3 to first obtain migration examples to feed into LASE. Furthermore, LASE does not automatically determine a match between examples and must be manually pointed to paired examples through manually edited configuration files, which we had to produce for each example set. Due to the high degree of manual effort when manually producing configuration files for LASE, we chose a subset of 10 of the 32 apps used for RQ2 to compare with LASE. The apps were selected at random, and contained a total of 17 migration possibilities comprised of 13 distinct APIs. The results can be seen in Table 19.

LASE can build a refactoring map for 15 (88%) migrations compared to 17/17 for A3. However, only 1/17 migrations were automatically mapped to migration points in the apps and subsequently applied to the apps. LASE appears to be highly dependent on the quality of the example and their similarity to the app code when compared to A3. We also find that LASE highly depends on complete matches in method AST. For example, if app code is inside an *if* statement or *try* block when the example code is not, if app code is instantiated differently than example code, or part of a method argument chain is in a different order to the example code, the AST sub-trees of the code will appear to be different. The differences in the sub-trees may lead to mis-detections between the primary example and the app code by LASE. We believe that since LASE was not designed for our use cases, it is intuitive that LASE is not optimal for our API migration task. On the other hand, A3 is designed to mine examples from the online code example, and hence is less sensitive to factors such as as complete AST-alignment and therefore can more often provide migration links when examples are loosely aligned.

We conducted four semi-structured interviews concerning three open source apps

from FDroid (*Antennapod, K9Mail, and Wikipedia*). We presented each developer with migration candidates from the app they were familiar with. We also allowed them to search the web, and ask any clarifying questions if they had any. We then provided one example of a migration of the same API call done in another app, mined by our approach. We asked four developers the following questions:

1. "Do you think our approach correctly identified the migration candidate?"
2. "How confident would you be when migrating this API?"
3. "How do you feel about the examples provided by our approach to help you migrate?"
4. "Do you think there are limitations or potential improvements to our approach?"

In all cases, the developers said that we had indeed found useful migration candidates. The migrations were judged to be easy, but time consuming. In all cases the developers said that our examples were *"extremely useful"* or *"very very useful"*, and that they would reduce the time to complete a migration from *"a few minutes, down to a few seconds"*.

One developer said that *"[...] where a new parameter is introduced it's hard to tell if the default value will do the trick, so I would likely have to do a decent amount of research before I felt secure in my choice [...]"*.

One developer identified a potential improvement when providing before/after examples rather than a complete migration, *"When you provide before/after files for a migration it would be nice to have a quick summary of the migration, like did you add a parameter? Otherwise I have to look at the documentation to make sure I'm looking at the right thing, and takes a little bit of extra time."*.

In one case a developer said that, *"Small examples are nice, but sometimes you want more context, so maybe having the whole file is better"*, on the other hand another developer said that, *"I prefer when I can only see the migration and a little surrounding code, if there's too much code, it's harder to see exactly what's going on."*. From these interviews we believe that there is an element of developer preference when looking at examples, and perhaps future research can look at the kinds of examples developers like, and create tools that can determine how much of data-flow and control flow is shown based on developer preference.

Table 20: Results of A3 user-study: comparing the time needed to migrate Android API usage examples (measured in seconds) with help from A3 and location-based API migration tools [52, 296].

Example	Avg. time w/o. A3	Avg. time w. A3	Time Improvement (%)	Avg. example usefulness
1	304.9	266.8	12.5	4.2
2	150.0	115.8	22.8	4.5
3	265.5	174.6	34.2	4.8
4	572.8	372.6	34.9	3.9
5	152.4	129.9	14.8	4.8
6	179.8	81.4	54.7	3.8
Total:	270.9	190.2	29.0	4.3

A3 can provide API migration assistance in 71/80 cases. We can automatically generate 14 faultless migrations, 21 migrations with minor code changes, and 36 migrations with useful guidance to developers. The effort needed to post-modify our generated API migration is low, an average of 3.65 tokens require modification.

RQ3: How much time can our approach save when migrating APIs?

In this RQ, we present the design and results of a user study involving the assistance provided by our approach. We conduct the user study to evaluate the usefulness of migration suggestions provided by our approach. The user study involves 15 participants (6 professional developers and 9 graduate students). In particular, we compare the time used for API migration by using our approach, and by only using location-based API migration tools, such as SEMDIFF [52], and AURA [296]. We do not compare to LASE [165] in this section since LASE cannot identify migration cites or examples without developer input. Table 21 shows the major differences between our approach and prior studies. Therefore, comparing A3, which automatically identifies a migration location, and automatically identifies potential migration examples, to LASE which can only migrate manually identified migration locations with manually identified examples, would not be a fair comparison of time saved. Rather, we seek to determine how much time can be saved by an approach that can automatically find migration examples and migration candidates when compared to approaches that

only identify migration candidates.

We extracted six API migration tasks from the out-of-date Android API uses from FDroid projects. Each participant was given three tasks with the help of migration suggestions provided by A3, and three other examples with the help of location-based API migration tools, such as SEMDIFF [52], and AURA [296]. We randomized the order of the tasks for each participant. These tasks were part of the dataset used in RQ1 and RQ2 and had been detected by A3 as APIs which needed migration. The tasks were chosen randomly from our sample to avoid bias and we manually ensured that each task did need migration. We used the approach presented in Section 4.3.2 to obtain a migration suggestion for each of the six tasks.

Each participant was given six source files that presented code with old versions of the Android API. The participants were told that they should attempt to modify the source code in each task to migrate to the latest version of the Android API. For the tasks that receive the help by A3, we provide the code that is generated by A3 after migration. For the tasks that only received the help from location-based API migration tools, the participants were informed of *which* API call to migrate and *what* is the new version of the API to migrate. We also provide the hyper-links to the Android developer website pages necessary to understand the API calls and their migrations were also given to the participants. We note that we did not directly run SEMDIFF [52] or AURA [296] to obtain above information, but directly provide the ground truth information to the participants, as if SEMDIFF [52] or AURA [296] generate perfect results. Furthermore, the participants were told that we provided potential solutions in the form of migration suggestions for some examples, and that they should attempt to use them if they could. This was done to minimize the noise in the measured time from other activities to concentrate on the migration of API calls themselves.

Related Work	Novelty provided by A3
CatchUp! [100]	Our approach does not require recording of API modifications.
SemDiff & AURA [52, 296]	Our approach uses API usage examples rather than internal API modifications to obtain migration patterns.
EXAMPLORE [91]	Our approach automatically obtains and applies relevant example code as patterns for migration.
Sydit & LASE [164, 165]	Our approach automatically obtains matched code examples, does not require fully compilable projects.
LibSync [191]	Our approach gives fully migrated code. and does not require fully compilable projects.

Table 21: Main differences and novelty provided by the A3 approach when compared to related work.

The participants were timed to determine how long each migration took. They

were also asked to rank the usefulness of the migration examples whenever possible. The rank is on a scale of 1 to 5, where 1 was considered as useless, and 5 was considered extremely useful. The results of our user study are presented in Table 20.

Overall, our approach provides an average time improvement of 29% with a p-value of 0.015 in a two-tailed Mann-Whitney U test. Professional developers improved by 45.9% while graduate students by 22.7%. We have therefore shown that automatically providing migration examples to users using a technique like A3 can improve migration times. As shown in Table 20, the developers involved with our user-study also found the assistance from A3 useful, ranking it an average of 4.3 out of 5 in usefulness. Therefore, our approach not only provides migration aid that can reduce migration time, but the examples provided are also judged as useful by developers.

Our approach provides, on average, a 29% improvement in API migration speed compared with location-based API migration tools. Users ranked the help provided by A3 an average of 4.3 out of 5 on a usefulness scale.

4.6 Threat to Validity

Construct validity. We assess the validity of our API migrations by building and running the apps as well as running the test suites of the migrated Android apps. Although we focus on the tests that exercise the migrated API calls, and attempt to exercise as much functionality as possible when running the apps, it is still possible that defects introduced by the migration are not identified by our tests. User studies and interviews with developers may complement the evaluation of our approach. In our study, there are still cases where our approach cannot migrate faultlessly. Although our approach can provide migration guidelines to developers, as an early attempt of this line of research, our approach can be further complemented by other techniques such as code completion to achieve better assistance in API migration. Furthermore, we concentrate on the majority of cases through file level migrations. If a large number of migrations occur across multiple files, our approach is not currently able to help.

External validity. Since this entire study was tested on the Java API of the Android ecosystem, it is possible that the findings in this chapter will not generalize to other programming languages. However, while it is true that the approach presented in this chapter was tested specifically on a Java based API, all of the approaches are

built upon assumptions that are true in other popular programming languages such as C#.

Internal validity. Our findings are based on the Android project and code examples mined and produced for its API. It is possible that we only found a subset of all migrations. It is also possible for the time gap between the release of new API and the update to examples to be larger in other sources. We attempted to mitigate these threats through mining official samples, open source projects, and having participants produce examples for frequently used APIs. We found that Google Samples updated deprecated API as soon as one month after the release of a new API version, which should allow developers to regularly update their apps. Our participant created examples were new and useful API migration examples, showing that the premise of using examples to help automate API migrations is functional and likely dependent on the sample size of examples.

4.7 Chapter Summary

In this chapter, we proposed an approach that assists developers with Android API migrations by learning API method call migration patterns from code examples mined directly from available code repositories. We evaluate our approach by applying automated API migrations to 32 open-source Android apps from FDroid and through a user-study. We find that our approach can automatically extract API method call migration patterns from both public code example and manually produced API examples that are created with minimal effort. By learning API migration patterns from these examples, our approach can provide either automatically generated API migrations or provide useful information to guide the migrations. Our user-study showed that the examples provided by our approach allow users to migrate Android APIs, on average, 29% faster and are seen as useful by developers, who ranked them an average usefulness of 4.3 out of 5.

This chapter makes the following contributions:

- We propose a novel approach that learns Android API migration patterns from code examples taken directly from available code repositories.
- Our novel approach can automatically assist in Android API migration based on the learned API migration patterns.

- We produce a user study and conduct semi-structured interviews that conclusively shows that migration examples are both desirable and useful to developers.

Our approach illustrates the rich and valuable information in code examples that can be leveraged in API related software engineering tasks. In particular we show that it is possible to use existing public code repository data to reduce the knowledge gaps between API users and API developers by using knowledgeable API user data (either directly from the Android development team, or from standard Android API users) to help with Android API method call migrations.

Chapter 5

Improving Misuse Detection Approaches

API migration is not the only issue faced by API users that can benefit from a reduction of knowledge gaps between API users and developers. If API users do not adequately understand an API, they can misuse it, to their detriment. Tools do exist to help API users uncover API misuses in their code. However, without a varied dataset of API usage examples, it is challenging for the example-based API misuse detectors to differentiate between infrequent but correct API usages and API misuses. Such mistakes lead to false positives in the API misuse detection results, which was reported in a recent study as a major limitation of the state-of-the-art. To tackle this challenge and reduce another knowledge gap between API users and developers, we first undertake a qualitative study of 384 API misuses randomly selected after using a state-of-the-art misuse detection tool on open-source software projects. We find that around one third of the false-positives are due to missing alternative correct API usage examples. Based on the knowledge gained from the qualitative study, we uncover five patterns to generate artificial examples for complementing existing API usage examples and avoid false API misuse detection.

To evaluate the usefulness of the generated artificial examples, we apply a state-of-the-art example-based API misuse detector on 50 open-source Java projects. We find that our artificial examples can complement the existing API usage examples by preventing the detection of 55 falsely detected API misuses. Furthermore, we use a state-of-the-art experiment dataset with an automated API misuse detection

benchmark (MUBench), to evaluate the impact of generated artificial examples on recall. We find that the API misuse detector covers the same true positive results with and without the artificial example, i.e., obtains the same recall of 94.7%. Our findings highlight the potential for improving API misuse detection by pattern-guided source code transformation techniques.

5.1 Introduction

Due to their ubiquity, determining how APIs are being used and misused is an important task in software development [23, 136, 142, 309]. Indeed, prior research has investigated how to improve API usability for API users [112,156,181,257]. Even with the existence of various studies and tools to improve API usability [84, 91, 185, 232], APIs still suffer from misuses [5]. APIs provide interfaces to existing functionality, these interfaces can be misunderstood and make it difficult for users to determine the correct way to invoke the underlying functionality [102, 229]. While API recommendation tools can provide a prescriptive way to address the misuse problem, they cannot address cases where a misuse already exists in a code base. API misuse detectors have therefore been created to uncover cases where APIs were used in potentially incorrect ways [5].

API misuse detectors, particularly those that employ API usage examples to uncover potential misuses, are at the mercy of the numbers of examples per API usage. A lack of usage examples was recently reported as one of the biggest challenges in API misuse detection [5]. In particular, uncommon API usages and alternative correct API usages have been found to make up 53.5% of false positive misuses [5]. An obvious solution to reduce the incidence false positives is to have a greater diversity of correct API usages examples. To achieve this goal, recent API misuse detectors, such as MuDetect, mine multiple projects to collect API usage examples [4], however the resulting false positive rate still has room of improvement [138].

In this chapter, we examine the challenge of missing correct API usage examples from a different perspective. Instead of mining source code from more projects to obtain more examples, we propose to generate artificial examples based on the existing correct API usage examples. The overview of our study is shown in Figure 17. We first undertake a qualitative study of API misuses identified by a state-of-the-art API

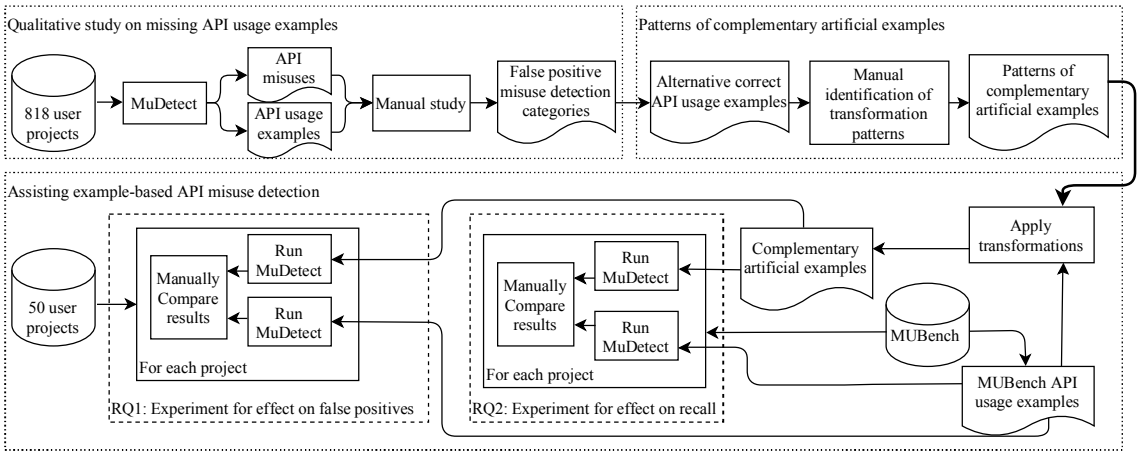


Figure 17: Overview of chapter 5 study setup, data collection, and experiments

misuse detector *MuDetect* in 818 open source projects. Through our study, we aim to discover patterns of opportunities where adding more API usage examples may reduce the false positive detection of API misuses. Thus, we identify five patterns to generate artificial API usage examples. Such artificial examples can then complement the existing API usage examples used in API misuse detection.

We evaluate the usefulness of these artificial examples using projects from another set of 50 open source projects and the MUBench dataset. Our evaluation shows that all five of our patterns can generate relevant examples that increase the precision of static API misuse detection approaches. Thus, we could eliminate 55 false-positives from the API misuse detection. In addition, through a pre-designed experiment in MUBench, we find that the artificial examples do not reduce true API misuse detection by the API misuse detector, i.e., the recall remains at 94.7%. The contributions of this chapter are:

- This chapter tackles the challenge of missing correct API usage examples from a different direction from prior research.
- Through a qualitative study, we identify five patterns of alternative correct API usages which can be used to generate artificial API usage examples.
- The artificial API usage examples can complement existing API usage examples to reduce the false positive detection of API misuses while keeping all true positives.

Our findings highlight the potential of generalizing correct and incorrect API usages based on pattern-guided source code transformations. We show that it is possible to use existing knowledge contained in public code repositories to increase the precision of API misuse detection approaches to reduce the knowledge gaps between API users and developers.

Chapter organization. Section 5.2 presents the background of example-based API misuse detection and a motivating example for our study. Section 5.3 presents our qualitative study on the false positive examples in API misuse detection. Section 5.4 presents the five patterns that can be used to generate artificial examples in order to complement the existing ones. Section 5.5 evaluates the usefulness of the generated complementary artificial examples. Section 5.6 discusses the threats to validity. Finally, Section 5.7 summarizes the chapter.

5.2 Background and a Motivating Example

We present the background of our study as well as an example to motivate our study.

5.2.1 Background: Example-based API misuse detection

Studies have demonstrated the advantages of example-based static misuse detection [5, 200]. While rule-based or constraint-based API misuse detection requires the existence of vetted knowledge of an API to codify usage rules that can then be used to detect misuses, example based detectors can rely on existing API usages to extract the knowledge needed for their detection [5, 200, 224].

Example-based static API-misuse detectors, such as MuDetect [4], extract real usages of APIs to indentify misuses. They are dependent on the samples of correct API usages, and their sizes. In a systematic evaluation of static API-misuse detectors, it was determined that over 53.5% of false positives of misuse detectors are due to uncommon or alternative correct ways to use an API [5]. This occurs even in cases where API-misuse detectors are trained in a cross-project setting, which provides more examples [4].

MuDetect is the state-of-the-art in example-based API-misuse detection, by default, it uses a minimum-pattern-support to allow examples with a minimum number

of examples to qualify as potential API usage examples. Potential API usage examples are greedily explored and clustered according to isomorphic pattern candidate extensions [4]. While clustering and extending example candidates, code semantics are also observed, in a graph form, for data and control nodes that could have side-effects or are oddly linked to the API example (e.g., only linked because of usage order, but not actually linked through any control action). Only if all nodes successfully pass through its heuristics does MuDetect consider a potential API usage example. These heuristics were designed in an attempt to prevent flagging uncommon usages as misuses [4]. In spite of these safeguards, false positive detection still occurs [4].

False positives are particularly hurtful to API misuse detectors by causing an over-reporting of misuses, which in-turn can overwhelm the users of these misuse detectors. Based on the 33.0% precision of state-of-the-art approaches such as MuDetect [4], it can be understood that example-based static API-misuse detectors would benefit from new ways to augment their sample of API usage examples.

5.2.2 A motivating example

Prior research suggests that a majority of false positive API misuse detection is due to a lack of less frequent API usage examples [5]. However, little is known about the nature of less frequent API usage examples. By observing falsely detected API misuses perhaps it is possible to determine some patterns of less frequent API usages and find new ways to help reduce the mistakes in API misuse detection. For example, in Figure 18 we can see an example of a falsely detected API misuse and the API usage example that was used for its detection. Based on the example presented, we can see that the API misuse detector falsely detects that it is a mistake to have the *put* API method in the *if* block rather than have it in a missing *else* block as it is in the API usage example used for detection. The API misuse detection tool does not recognize that the conditional statement that determined whether the key is already contained in the map has been inverted in the wrongly detected API misuse.

This lack of knowledge stems from a lack of varied API usage examples. Current state-of-the-art approaches attempt to remedy this problem by mining API usage examples from large inter-project data sources [4]. However, although this does improve the performance of API misuse detectors, the problem persists. If patterns

```
public void fieldEofResponse(byte[] header, List<byte[]> fields, byte[] eof) {
    //… extra code was removed for brevity
 Map<String, ColMeta> columToIndx = new HashMap<String, ColMeta>(fieldCount);
 for (int i = 0, len = fieldCount; i < len; ++i) {
        //… fieldName instantiation was removed for brevity
  if (columToIndx != null && !columToIndx.containsKey(fieldName)) {
   //… extra code was removed for brevity
   columToIndx.put(fieldName, colMeta);
  }
 }}
```

(a) An API usage falsely detected as an API misuse

```
public void pattern(Map<String, Object> m, String key, Object value) {
    //… extra code was removed for brevity
 if (m.containsKey(key)) {
    //…
 } else {
    //…
   m.put(key, value);
}}
```

(b) The API usage example used for API misuse detection

Figure 18: An example of a falsely detected API misuse.

of common cases of wrongly detected API usage examples exist, such examples can potentially be directly generated based on the patterns without needing to search for all possible API usages. For example, to address the issue in Figure 18, if a complementary example existed in which the *put* method call was located in an *if* statement with inverted logic, i.e., *!containsKey()*, the API usage shown in Figure 18a would not be falsely detected.

Therefore, in the next section we conduct a qualitative study in an effort to uncover patterns of missing API usage examples.

5.3 Qualitative Study on Missing API Usage Examples

In this section, we first conduct a qualitative study to gain understanding of the missing correct API usage examples.

5.3.1 Qualitative study setup

Although work has been done to determine the caveats and problems with existing misuse detection techniques, work remains to be done to determine strategies to handle infrequent API usages and alternative usages for the same API [5]. We therefore seek to systematically determine patterns of alternative API usages and how to leverage them to reduce false-positives in API misuse detection. We first present the projects and tools used to conduct our preliminary study.

Subject projects. We use a readily available dataset [136] from a recent API research work that includes 3,099 Java projects available on Github. Although the dataset was originally assembled to study five open-source Java APIs (Guava, Hibernate-ORM, Jackson, JUnit, and Log4j), the projects in this dataset are not limited to using only these five APIs and therefore present a rich source of varied API usage. We intentionally do not select an existing API misuse dataset (like MUBench) for this qualitative study in order to avoid the bias (positive or negative) of existing knowledge in the benchmarks of API usage examples.

To conduct our experiments, we selected a sub-sample of 1,000 projects, randomly selected from the original sample. Out of these 1,000 projects, we discovered that 132 were incompatible with MuDetect due to compilation errors. Our final sample size was therefore 868 projects. We reserved 50 of those projects for our final evaluation and the rest was used for our manual study. The names and download links for the projects used for this study, as well as the results of our experiments can be found in our replication package[1].

API misuse detection. To study the cases where valid API examples are missing, we use an automated tool to detect API misuses, and further examine the false positives in the detection results. We opt to use MuDetect [4] as an API misuse detector because it is a vetted API misuse detector that has shown state-of-the-art

[1]the replication package can be found at: https://github.com/senseconcordia/API-Workarounds

results. More importantly, MuDetect has the ability to uncover API usage examples against which it can measure potential API misuses [4]. This automatic mining of frequent API usage examples allows us to leverage the large scale of open-source repositories to mine usage examples from a wide variety of APIs and obtain varied samples of examples of API usages that could be qualified as odd or misused.

We ran MuDetect in its intra-project mode. We did not opt for its inter-project mode because running the inter-project mode on a sample of 818 projects is prohibitively time consuming due to the explosion of misuse patterns that occurs. Furthermore, using the intra-project mode allows us to obtain a "worst-case" real usage scenario that serves our goal of studying missing API usage examples.

MuDetect uses multiple heuristics to identify frequent API usages. The first heuristic is based on the frequency of API usage patterns. MuDetect has a minimum-pattern-support variable which allows any API with more usages than the set threshold to qualify as a potential API usage pattern. For the automatic extraction of API usage examples used in our qualitative study, we set the minimum-pattern-support to its default value to allow patterns with a minimum number of examples to qualify as potential API usage patterns. We used this value because it was successfully used in the MUBench dataset for its evaluation [4]. Furthermore, a lower threshold would allow for more false positive API usage patterns and increase the already non-trivial detection time.

MuDetect saves API call information on a per-misuse basis. The location (i.e., file, line number, calling method) of API misuses as well as the locations of API usage examples used for detection are recorded as part of the MuDetect tool process. We therefore use this information to manually observe real instances of potential misuses.

5.3.2 Qualitative study process

We obtain 206,302 potential API misuses. To understand the API misuses that are detected due to the lack of API usage examples, three reviewers conducted a manual study on a statistically representative sample of 384 detected API misuses (with a 95% confidence level and a 5% confidence interval). Our final goal of the quantitative study is to uncover patterns of missing API usage examples that cause false positive API misuse detection.

Our study process makes use of Krippendorff's α, which can be used to determine

a quantitative agreement between coders of typically unstructured data [132].

Krippendorff's α provides a value between zero and one to indicate the observed disagreement between coders. If coders agree perfectly then α=1. In the case where coders present an agreement equivalent to random chance then α=0. Therefore, reliable data is represented as an $\alpha \rightarrow 1$, and should be far from α=0. Krippendorff's α takes the form of:

$$\alpha = 1 - \frac{D_o}{D_e}$$

where D_o is the observed disagreement between coders and D_e is the disagreement expected by chance. Details related to the calculation of Krippendorff's α can be found in [131].

In this step, we first study the false positives in the detection results to identify the ones that may be caused by missing API usage examples. For each detected API misuse, the reviewers also observe five API usage examples that were leveraged by MuDetect to detect the misuse. The five API usage examples can help the reviewers understand why an API misuse was detected. The manual study includes four steps.

Step 1. We first start by manually examining a sample of 174 detected API misuses. Each of the three reviewers was given 116 random potential misuses to categorize as they saw fit (i.e., open card sorting). Each misuse was examined and categorized by two of the three reviewers.

Step 2. Once all of the 174 misuses were categorized, the three reviewers discussed their categories and settled on a base classification schema.

Step 3. All reviewers reexamined their categorization results using the newly agreed schema. Once all of the 174 detected API misuses were classified according to the same schema, we measured the agreement ratio using Krippendorff's α [13, 131, 132]. The calculated agreement ratio was 0.744, i.e., a substantial agreement for consensus. Afterwards, all the three reviewers discuss the cases of disagreement, until final categorization of the 174 misuses were made.

Step 4. Due to the substantial agreement ratio achieved in the last step, a further sample of 210 misuses (for a total of 384 categorized potential misuses) was therefore manually classified by three reviewers without the need of overlapping, unless necessary (e.g., if a reviewer felt unsure about the classification). Whenever a reviewer believed that a new category was identified during this step, all three reviewers discussed the particular case.

5.3.3 Qualitative study results

After the four steps, we put the 384 detected API misuses into a total of four categories.

Alternative correct usage: *(108 instances)* This category is used to describe API usages that are similar to the ones in the API usage examples that were used for detection. However, although similar, these falsely detected API usages use some alternative means of working with the API that could potentially have been detected by an API usage example complementary to the ones used for detection. API usages categorized in this category are used for further inquiry into potential transformation patterns to create complementary artificial examples.

Different usage scenarios: *(155 instances)* This category is used to describe API usages that were used in different scenarios. Contrarily to usages categorized as *alternative correct usages*, we could not identify how these usages could have been detected with complementary API usage examples, since these APIs are used to service different proposes. Therefore, these API usages require completely different API usage examples to the existing ones.

Correct misuse: *(13 instances)* Some of the examples that were selected for manual review were correctly identified by MuDetect as misuses, therefore we categorized these examples as such.

Not sure: *(108 instances)* If the reviewers could not identify why an API usage was targeted as an API misuse by MuDetect or whether the API usage was a misuse or not, we categorized the API usage as "Not sure".

5.3.4 Summary of the qualitative study results.

Around one third of the overall results of our qualitative study are "alternative correct usages". These manually identified alternate API usages that were falsely identified as API misuses present opportunities for us to identify which alternate correct usages cause confusion in API misuse detection. The found prevalence of alternative correct usages provides an opportunity to identify general patterns of alternative correct API usages. Through these general patterns we can transform the existing frequent API usage examples into less frequent alternative correct examples, in order to address the challenging of missing API usage examples. Such examples would later help reduce the rate at which those API usages are falsely detected as

API misuses.

5.4 Patterns of Complementary Artificial Examples

Section 5.3 shows that a considerable amount of API misuse detection results are actually due to missing correct API usage examples that represent the alternative correct usages of an API. If more usage examples of these correct API usages were available to complement the existing examples, they may significantly reduce the false positives in API misuse detection.

Therefore, in this section, all reviewers discuss each API misuse detection results that were classified as "alternative correct usage" in the qualitative study (cf. Section 5.3), to uncover patterns of the needed complementary examples that can be used to reduce the false positive detection results. For each of the patterns, the authors further discuss whether an artificial example can be automatically generated based on transforming existing API usage examples. The discovered pattern can later be used to generate artificial API usage examples to complement the existing examples to identify correct API usages and reduce falsely detected misuses. In total, we discover five such patterns.

In the rest of this section, we discuss each of our five manually identified patterns in the following template:

Description: Description to the pattern of complementary API usage examples.

Example: Discussion of a concrete example that is presented in Table 22.

Detection strategy: Our strategy to detect possible API usage example candidates for the pattern.

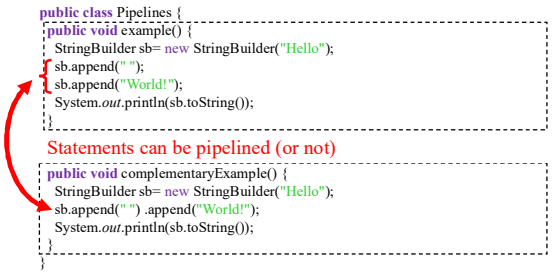
Transformation approach: Our approach to generating artificial API usage examples based on the transformation of an existing API usage example.

Pattern 1: Pipelines.

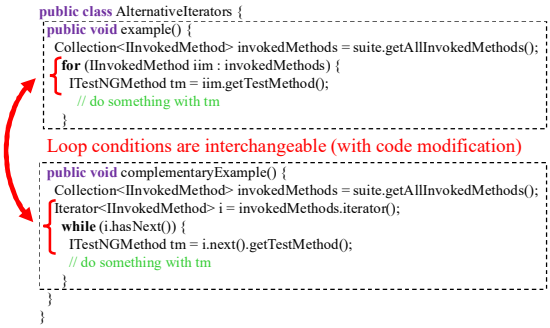
Description. This pattern is built primarily around APIs that can be used in stages on the same object, also known as the pipeline pattern [60]. Using the pipeline pattern requires that an API method return the same object type as the calling object type. This pattern is particularly useful because developers sometimes pipeline a few

Table 22: Patterns of complementary artificial API usage examples.

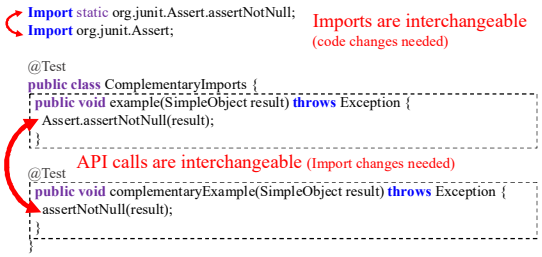
Pattern 1: Pipelines



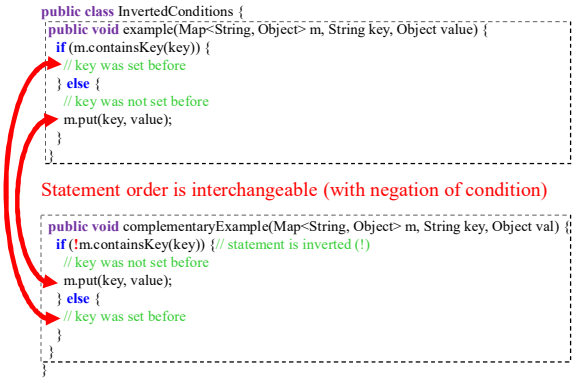
Pattern 2: Alternative iterators



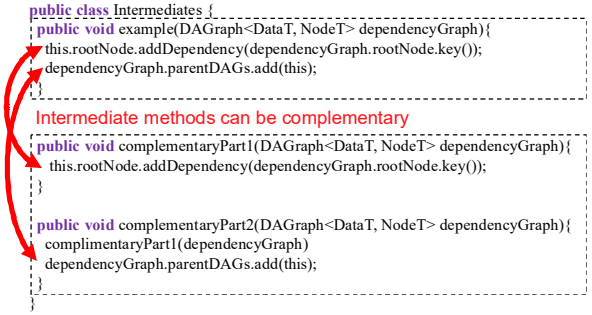
Pattern 3: Complementary imports



Pattern 4: Inverted conditions



Pattern 5: Intermediates



API call stages, and sometimes they use individual stages, one at a time.

Example. As shown in the example in Table 22 (Pattern 1), a string builder can be used to append new string to the end of an existing string. The string builder can call the API method *append* multiple times, separately, to append different strings to its end. On the other hand, the API method *append* returns a reference to the original string builder to enable the method to be called in a pipeline. Hence, it is semantically equivalent to either call *append* twice in separate, or in a pipeline. Therefore, it is possible to artificially generate a complementary API usage example, if such an API is called repetitively either separately or in a pipeline.

We would like to mention that the number of strings that can be appended is not defined ahead of time, and therefore cannot be inferred. In our example only two strings (" ", and "World!") are being appended for brevity. However, one could append many more, or none at all, while still being a correct API usage. However, since we cannot produce all possible numbers that an API is repetitively called, we opt to only generate the complementary example with the same number of calls.

Detection strategy. The *Pipelines* pattern requires an API that is called on, and returns, the same object. This can be ascertained via abstract syntax tree (AST) analysis with source bindings or through inference if a collapsed form pipeline is available (e.g., the *sb.append("").append("World!");* in the `complementaryExample()` method in Table 22). However, there are exceptions to this rule, for example Java Streams present a pipeline-like pattern, but they cannot be broken up into different stages. Java Streams stages must be done in a single pipeline, otherwise a new Stream must be created [47].

In some cases it might not be possible to fully collapse the stages of otherwise pipeline-able code. In our example of Pattern 1 in Table 22, if the string "World!" was assigned dynamically after having started the *StringBuilder* (e.g., by requesting user input). In that case, we would have to determine if the dynamic assignment could be moved outside of the pipeline pattern. In cases where intermediate instantiating of variables occurs between otherwise pipeline-able API calls, we must determine if moving those intermediate variables would change the semantics of the method. This analysis can be done by using control-flow and data-flow analysis. However, even in cases of unmovable intermediate variables, if there are many pipeline-able stages, it might still be possible to partially collapse some stages (e.g., all stages

before an intermediate variable appears), which can be done to produce a possible complementary correct usage.

Transformation approach. If starting from the top method (*example()*), the stages of the pipeline can be collapsed after the first API call (in this case *.append("")*). The pipeline stages can then be considered as a single statement. The reverse is also possible. It is possible to transform the pipeline form presented in *complementaryExample()* into its non-collapsed version in *example()*. This can be done by determining the return type of a pipeline stage (in this case *.append("")*). If the return type matches the type of the original object (in this case *StringBuilder*), then we can separate the stage by calling it on the original object in a new statement (semi-colons must be adjusted). The original order should be preserved.

Pattern 2: Alternative iterators.

Description. This pattern arises from the different loops that are allowed a programming language (e.g., Java in our study). During our manual study we noticed that some examples of API usages were being falsely identified as API misuses because their control-flow was directed by different types of loops (e.g., *for* loop, *while* loop and *forcach* loop). We are particularly interested in loops that can made into other types without changing the behavior of the program, or the API call under inspection.

Example. Table 22 (Pattern 2) presents an example of a *for* loop that was transformed into an equivalent *while* loop using an iterator. In this example, the *getTestMethod()* API call can still be used to obtain the same effect. However, the syntax of the context of the API call has changed. Different programmers appear to have different preferences for using different types of loops. Perhaps the legibility of code differs for developers based on their proficiency with various loops. The fact that different developers can use different loops to achieve the same effect is important. Developers seldom work alone on software projects, and therefore it is possible for different types of loops to appear with the same API call. If one type of loop is not frequent enough, it may be mistaken as a misuse when compared to examples with other loop types.

Detection strategy. The *Alternative iterators* pattern requires that an API call be affected by a loop (either by data or control-flow). If an API call is inside a loop but is not affected by the loop, it should be possible to construct a sub-graph of the usage

that can cover any loop scenario [5]. In such cases, the loop itself does not impact the correctness of the API call. Furthermore, for us to produce a transformation it should be possible to transform a loop into another type only by referring to method calls available in the Java standard library. If an example requires code modification outside of the Java standard library, we do not attempt the transformation to limit the introduction of defects.

Transformation approach. As shown in Pattern 2 in Table 22, we can easily transform a *foreach* Java loop into a *while* loop by introducing an intermediate iterator and the *hasNext* and *next* methods on this iterator. The same can be achieved in reverse. Similarly, it is also possible to transform a standard Java *for* loop into either of these types with the introduction of some intermediate variables that can be inferred from bindings in the existing code example. If we cannot infer all bindings, we do not attempt the transformation.

Pattern 3: Complementary imports.

Description. In our manual study, we uncovered cases where developers had imported static API methods and used them directly. In other cases, the same API was statically called from the owner class.

Example. This can be seen in Pattern 3 of Table 22 where the *assertNotNull* API call occurs in both example methods. However, one of them is called explicitly by the *Assert* class; while the other one relies on the *import* statement. If one of these usages is missing from the examples, it is possible to mistake the missing one as a misuse.

Detection strategy. To detect this pattern, an API call must use a static method. Furthermore, it should be possible to import this static method independently from the class itself, as shown in the *Import static org.junit.Assert.assertNotNull* statement in Pattern 3 of Table 22. In-depth binding analysis could be used to detect this pattern. However, in-depth binding analysis can be computationally expensive, or even impossible in malformed projects, and could slow down already non-trivial API misuse detection times if done for every occurrence. To address this, we use a heuristic based on typical Java naming convention where the name of a class should start with a capital letter. Therefore, we can obtain a list of imported static method calls in the *import* statements and the calls to the static method calls in the source

code.

Transformation approach. As shown in the method in Pattern 3 of Table 22, the *Assert.* class call in the *example* method can be removed as shown in the *complementaryExample* method. The corresponding static import to the target API method must be determined and added to allow the new syntax. The reverse is also possible. If a static method import is in use with a direct import, the corresponding import to the target class must be determined and added, and the static method must be modified to be called on the imported class.

Pattern 4: Inverted conditions.

Description. Similarly to our *Alternative iterators* pattern, different developers appear to use different orders for their conditional statements. Although good practices [160] suggest using logic and naming conventions that make sense with respect to chosen names and reduce the number of double negatives (e.g., *!doesNotContain* instead of *contains*), it is still possible for situations to arise where two (or more) equivalent correct code expressions exist.

Example. In the Pattern 4 example in Table 22, we present a simplified example of this pattern. In this example we can see that reversing the logic inside the conditional *if* statement changes the logic of the overall method. The *put* API call is therefore moved from the *if* block to the *else* block when the condition is inverted. As shown in the example, this pattern can work in either direction.

Detection Strategy. First, this pattern requires the use of a conditional statement (*if* statement) to gatekeep API usage. In Pattern 4 of Table 22 this API call is the *put* method. Currently we only consider cases with single *if/else* statements, we do not apply the patter if any *else if* statements are involved. Afterwards, we investigate the possibility of switching the order of conditional statements. Similarly to Patter 3, it is less computationally expensive to determine an equivalent loop once for a known API usage example to create a complementary usage example, than it is to determine the equivalence of every loop analyzed by an API misuse detector.

We have also seen and considered cases where a complementary example could be produced by exchanging an *if/else* statement with a *try/catch* clause. However, it was not readily apparent how this particular version of the pattern could be generalized safely, while remaining certain that we would not introduce undesirable side-effects.

However, we present this strategy here since we did find instances of these false positive API misuse identification in our manual study.

Transformation approach. The pattern first requires the inversion of the conditional logic in an *if* statement. After the conditional logic has been inverted, the functionality that was originally in the *if* block can be transferred to the *else* block, and vise-versa. If multiple conditions are present in the if statement care must be taken to either invert the all conditions separately without fail or invert the complete statement as one piece. The inversion of the logic inside the *if* statement in its simplest form can stem either from the removal or addition of the "not" operator (i.e.,!).

Pattern 5: Intermediates.

Description. This pattern allows more flexibility in the expression of API usage examples by parameterizing intermediate functionalities. Existing functionality that is related to an API usage can be extracted to a new method and replaced with a method call.

Example. A simplified example is presented in Table 22 (Pattern 5) where the *complementaryPart1* method replaces functionality originally in the *example* method. This example is trivial since it does not reduce the number of statements in the *complementaryPart2* method. However, the transformation can be expanded to more complex forms that would apply some separation of concerns.

Detection strategy. This pattern requires the possibility of extracting functionality, abstracting this functionality to an intermediate method and either invoking it in the original method or introducing it to the original method as a parameter. We require that the control and data-flows of a method be separable before or after an API call. If the method statements are heavily coupled, this transformation cannot be applied.

Transformation approach. The control and data flows of the original method must then be analyzed to determine where a proper method extraction could occur. The transformation approach is similar to an *extract method* in refactoring [271]. However, there exist infinite possibilities to extract new intermediate methods from an existing program. Therefore, in our heuristics we only extract statements adjacent, i.e., right before or after, to the target API call. Once an intermediate method has been introduced, the original method must be modified to remove the functionality

that was extracted, and instead call the intermediate method, with its appropriate parameters.

5.5 Assisting Example-based API Misuse Detection

In this section, we evaluate the usefulness of our generated artificial API examples that are based on the five patterns discovered in Section 5.4. In particular, we perform an experiment that detect API misuses in open source projects, with and without the use of the artificial API usage examples. In the rest of this section, we present the subjects of the experiment, the experimental process, and the experimental results.

5.5.1 Experimental subjects

We use the existing API usage examples in the MUBench dataset [3] as the API usage examples for the API misuse detection tool. The MUBench dataset provides a baseline of known and vetted API misuses in a variety of real projects. To test our complementary examples, we specifically rely on already known API usage examples that have been used to identify misuses in the MUBench dataset. The MUBench dataset comes with a prepared benchmark named *FSE18-Extension* that contains 107 known misuses, each with a single known API usage example.

There may exist other API usage scenarios that the data in the MUBench dataset does not cover. Therefore, as mentioned in Section 5.3, we use 50 open source Java projects from a prior study that mined these projects from Github [136]. To avoid bias, we make sure that none of the 50 projects were included in our qualitative study where the patterns were discovered (cf. Section 5.3 and Section 5.4). All of these projects are used in prior API-related research and had at least 17 months of history and had an average of 26K lines of code (minimum: 1.4K, maximum: 225K). Due to the limited space, we cannot present the details of all 50 projects, the details can be found in our replication package identified in Section 5.3.

5.5.2 Experimental process

In particular, our experimental process is designed to answer two research questions:

- **RQ1:** Can artificial examples reduce false positives from the API misuse detection results?
- **RQ2:** Would artificial examples negatively impact the detection of any true API misuse?

Experimental process to answer RQ1. We obtain the existing API usage examples that are provided by MUBench. We then detect API misuses using the state-of-the-art API misuse detection tool, i.e., MuDetect, on our 50 open-source projects with the existing examples as a baseline. We then generate artificial API usage examples to complement the existing API usage examples from MUBench, based on the five patterns shown in Section 5.4. With the artificial examples, we rerun MuDetect on the 50 subject projects.

By default, MuDetect would consider every combination of examples to attempt to detect API misuses. In order to observe the value of the artificial examples in complementing existing examples, we therefore force MuDetect to consider our complementary artificial API usage examples immutably paired to the original API usage examples. Note that, we do not aim to use the artificial examples to replace the existing ones. Instead, the role of the artificial example is only for complementary purposes. Finally, we examine the results of both runs to answer the first research question.

Experimental process to answer RQ2. We would like to ensure that our complementary examples do not negatively impact the overall power of API misuse detectors. However, our experimental process for RQ1 cannot serve for this goal since there exists no ground truth on all of the true API misuses in the 50 open source projects. On the other hand, there exists a specially designed experiment, i.e., the *ex1* experiment in MUBench, which particularly serves the goal of calculating a recall upper bound for a given API misuse detectors with known API usage examples. We therefore run this experiment with only the original API usage examples. We then run the experiment again, having the existing examples complemented by the artificial examples. We compare our results to determine the effect of our complementary API usage examples on the recall of an API misuse detector. In particular, for each run, we follow the same evaluation approach as Amann et al. [5] and record a positive identification of an API misuse if any detection proves to be positive.

5.5.3 Results

All five of our patterns are applicable to complement the existing API usage examples in MUBench. Out of all 107 API usage examples in MUBench, only one example does not meet the detection strategy of any pattern. This API usage example was created for a synthetic Java survey, and contained a single statement composed of five Java stream stages. Since we strategically do not consider streams to avoid potential mistakes (cf. Pattern 1, Pipeline in Section 5.4), we were unable to generate any complementary API usage examples for this particular API usage example.

In particular, we were able to improve three existing API usage examples with complementary API usage examples using the *Pipelines* pattern, eight using the *Alternative iterators* pattern, 22 using the *Complementary imports* pattern, eight using the *Inverted conditions* pattern, and 65 using the *Intermediates* pattern. Although we were able to express all five of our patterns on this dataset, the *Intermediates* pattern stands out as the most popular pattern in this case. This dataset contained methods where further separation of concerns could be introduced, which allowed for a greater expression of our *Intermediates*. A different dataset could perhaps present different pattern frequencies. However, because our patterns were generated on a completely separate dataset, and yet all five of our patterns could be used in the MUBench dataset, we are confident that our patterns present potential for general complementing of existing API usage patterns.

The artificial examples can assist API misuse detection by reducing false positives. In total, we find 55 cases of API misuse detection results that were detected with the original examples but were not detected when complemented by our artificial examples. We then manually go through each of those 55 examples to determine if they were API misuses or not, and which of our transformation patterns was used to disable their detection. We find that all 55 API usages were correctly identified when using complementary artificial examples, and that they were originally mistakenly labelled as API misuses.

We find that 43/55 (81.8%) of the mistakenly identified API misuses that were corrected by our complementary artificial examples used some kind of conditional statements. This shows that it is particularly important to have a well-rounded sample of API usage examples that contain various types of conditional statements

because their detection appears sensitive to their format. The second most common mistakenly identified API usage type used iterators (6/55), this can also be used as a suggestion for future API misuse datasets to carefully consider various types of loops or use complementary artificial examples to enhance them. Finally, we report 3/55 wrongly identified API misuses with intermediate methods, 2/55 with pipelines, and 1/55 with static imports. Although these cases are less prevalent, we still encourage the use of their transformation patterns because they still provide a reduction in false positive.

Answer to RQ1: 55 falsely detected API misuses were prevented by complementary artificial examples. Complementary artificial API usage patterns can therefore successfully be used to reduce the incidence of false positive API misuse detection on real world projects.

The artificial examples do not prevent the detection of true API misuses. Due to build errors in the MUBench dataset we were able to obtain results for 95/107 API misuses in the dataset. MuDetect was able to successfully detect 90/95 of these misuses (94.7% recall) both with and without our complementary examples. All 90 correctly identified misuses were the same for both experiments (i.e., with and without our complementary examples). Therefore, we do not find any case where using complementary artificial API usage examples reduce the overall recall of API misuse detectors.

Answer to RQ2: Our findings on the MuBench dataset indicate that using complementary artificial API usage examples does not reduce the overall recall of API misuse detectors (94.7% recall with & w/o our complementary examples).

5.6 Threats to Validity

Construct validity. We do not claim to have found all API misuses, falsely detected API misuses, or API usage patterns pertaining to the APIs in this study. However, we believe that the projects and tools used in this work are adequate to produce results that give insight into the problem at hand. Although we diligently attempted to confirm the results presented in this chapter by searching application documentation, online forums, by using existing state-of-the-art tools, by labelling unsure findings as

such, and by using existing and vetted datasets it is still possible that some of the results presented were misidentified. We do not claim to be experts for any of the user applications that we studied nor for any of the APIs that we studied.

External validity. Since the API misuses, workarounds, and frequent pattern instances in this study were detected for Java APIs in Java user applications, it is possible that the findings in this chapter do not generalize to other programming languages. However, although the results presented in this chapter were obtained from Java APIs, the results were obtained by mining hundreds of user applications for API misuses without discriminating against any particular APIs. We therefore believe that although we cannot prove that our results generalize to other programming languages, the results presented should generalize to Java APIs.

Internal validity. The patterns presented to produce complementary artificial API usages, the suggestions presented for future API misuse detectors, and the findings from our qualitative study might not be fully indicative of API misuses and could present internal experiment bias. We attempted to mitigate these threats by having multiple reviewers for the API misuses that we presented in this work, and having these reviewers reach consensus on discussions pertaining to the patterns and suggestions that we present in this chapter. Furthermore, we use completely different samples to obtain and to test the patterns presented in this work. Although the sample size of our qualitative study is statistically significant (384), it is possible that our findings only generalize to the MuDetect tool. However, MuDetect uses a published and general misuse detection approach that has been shown to be the current-state-of-the-art in example based static API misuse detection. Therefore, we believe that our results can contribute to improving the current state-of-the-art.

5.7 Chapter Summary

In this chapter, we conducted a qualitative study on the falsely detected API misuses obtained by using a state-of-the-art example-based API misuse detection approach on a large sample of projects. By manually studying real examples of falsely detected API misuses, we uncover 108 cases of alternate but correct API usages that were falsely identified as API misuses. Through a manual investigation by three reviewers, we discover five patterns, which can be used to transform existing API usage examples

into artificial API usage examples. Such artificial examples can cover the knowledge gaps caused by a lack of diversified existing API usage examples. We provide detailed discussions and simplified examples to explain these five patterns, as well as our strategies to detect these patterns in the source code, and approaches to transform existing API usage examples with these patterns.

We evaluate the usefulness of the complementary artificial API usage examples through the use of 50 open-source Java projects and through the MUBench misuse benchmark. We find that using the artificial examples does not reduce the recall of API misuse detection but does allow for the removal of falsely identified API misuses. Our findings highlight the potential of generalizing API usage examples through pattern-guided source code transformations and reduce the dependence of example-based API misuse detection on haphazardly mining large samples of user projects. Our findings can be used to improve the precision of API misuse detection approaches and therefore reduce API user knowledge gaps that cause API misuses.

Part III

Aiding API Developers

Chapter 6

Guiding API Development by Using API Workarounds

Although issues such as API migration and API misuses show that API users suffer from gaps in their knowledge of APIs, API developers are not immune to knowledge gaps between them and API users. Knowing if and when API users are not fully satisfied with an API can allow API developers to improve their APIs and remain relevant to their users. APIs are not guaranteed to contain every desirable feature, nor are they immune to software defects. Therefore, API users will sometimes be faced with situations where a current API does not satisfy all of their requirements but migrating to another API is costly. In these cases, due to the lack of communication channels between API developers and users, API users may intentionally bypass an existing API after inquiring into workarounds for their API problems with online communities. This mechanism takes the API developer out of the conversation, potentially leaving API defects unreported and desirable API features undiscovered.

In this chapter we explore API workaround inquiries from API users on Stack Overflow. We uncover general reasons why API users inquire about API workarounds, and general solutions to API workaround requests. Furthermore, using workaround implementations in Stack Overflow answers, we develop three API workaround implementation patterns. We identify instances of these patterns in real-life open source projects and determine their value for API developers from their responses to feature requests based on the identified API workarounds.

6.1 Introduction

All too often, when users have issues with an API, for example needing a new feature or experiencing a run-time problem, users may choose to intentionally modify or bypass the API [30]. We define API workarounds as source code produced by API users, without official support from API developers, for the intentional modification or bypassing of official APIs. These workarounds allow API users to obtain their desired functionality quickly and without going through potentially arduous communication with API developers. However, the introduction of API workarounds presents a dilemma for API developers and users. On the one hand, because these workarounds are created by API users as temporary solutions, they become technical debt [216], endangering code quality and increasing future maintenance cost [306]. On the other hand, these workarounds may become a vehicle for the API developers to gain feedback from API users, to improve the APIs (e.g., fixing defects in the API).

We conduct an exploratory study of API workarounds requested and implemented by API users. To start our exploration, we manually examine 400 posts from Stack Overflow, where we found that API users request API workarounds for a variety of reasons, such as dependency issues, missing functionality, and runtime problems. These reasons illustrate inherent value for API developers to improve their APIs. Furthermore, we identified answers accepted by API users who request API workarounds. By studying these answers, we found that carrying out such API workarounds may not be a trivial task. In particular, a majority of API workaround solutions require previously unknown implementations to bypass the API.

To follow up on our exploratory study, we study workaround implementations that are suggested in the Stack Overflow posts, and we observe three generalized API workaround patterns. The knowledge contained in the implementation of these patterns in API user projects can help API developers improve their API by adding desirable unsupported features, fixing unexpected behavior, and improving backwards compatibility.

because the three API workaround patterns were uncovered using forum questions and answers, we seek to confirm their existence in real-life API user code and confirm their usefulness with API developers. Using five open-source APIs, we detect these three patterns of API workarounds in open-source GitHub projects. Finally, we submit and observe 12 feature requests to developers based on the API workarounds to

improve their APIs. Among these requests, five are already closed, and six more have been confirmed by API developers as defects or missing features.

Our study and findings highlight the value of studying the usage of APIs from API users as a means to bridge the knowledge gap between API developers and API users in order to assist in the development and maintenance of APIs.

Chapter Organization. Section 6.2 provides real examples of API workarounds requested on Stack Overflow to motivate this study. Section 6.3 contains a qualitative study on API workarounds. Section 6.4 presents three generalized API workaround patterns extracted from API workaround implementations in Stack Overflow answers. Section 6.5 presents an empirical study conducted to find and report instances of three API workaround patterns. Section 6.6 describes threats to the validity of this study. Finally, Section 6.7 summarizes the chapter.

6.2 Motivating Example

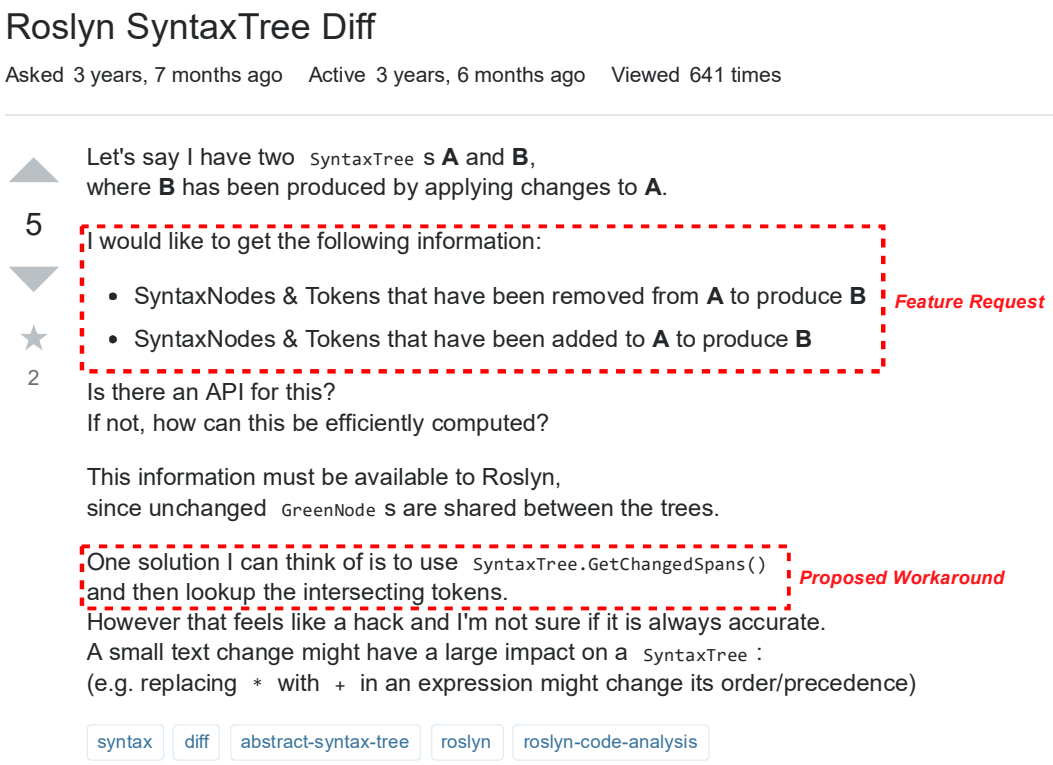


Figure 19: An example of a developer requesting access to data that exists in the Roslyn API but appears unaccessible

Figure 19 presents a question (Stack Overflow id: 34945023) in which the poster (API user) inquires about working around the Roslyn API. The API user requires access to data that appears to exist in the Roslyn API; while such data cannot be externally accessed by the API users. The API user provides a short example of their desired functionality, and asks if this functionality already exists in the API. The API user explains why they would like the feature, and why they believe the feature should already exist as part of the API. Finally, the API user provides a potential workaround solution to obtain the desired data, but still expresses a desire for direct support from the API.

The accepted answer post, presented in Figure 20, was provided by one of the API's developers. This post provides insight into a direct information exchange between an API user requesting an API workaround, and an API developer.

The API developer confirms that, the feature requested by the user is indeed available internally, but was not exposed to the public, it can be considered an implementation detail. The API developer claims that there is no specific reason for that API to be hidden. The API developer also mentions that they could not think of a motivating scenario for this feature.

From this example, we can see that: 1) Scenarios of how APIs are used by users may be unforeseen by API developers, 2) The effort needed and challenges encountered by API users to make API workarounds may not be trivial; while the requested feature/information may already exist internally in the API, or require much less effort for API developers to accomplish than the API users, and 3) API workarounds provide valuable information for API developers in order to understand the needs from the API users and to improve their APIs.

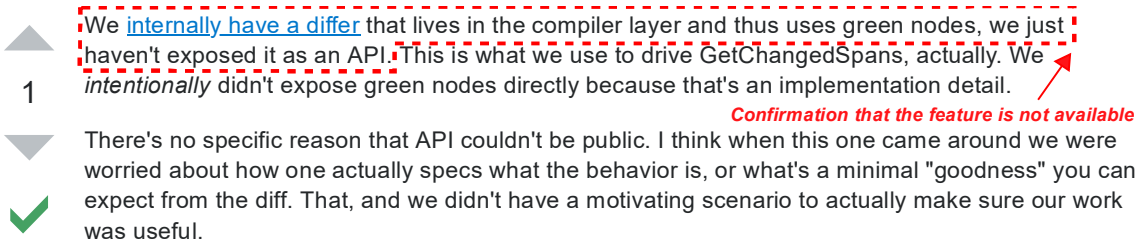


Figure 20: Example of an API developer answering an API workaround request for data that exists in the Roslyn API but appears unaccessible

This example shows that a disconnect can exist between API developers and users.

On the one hand, users sometimes prefer to use public forums to request functionality rather than having direct communication with the API developers. On the other hand, not all API inquiries lead to responses from API developers, who may miss these opportunities and fail to obtain outstanding sources of knowledge from their API users, which they could use to improve their API. Therefore, in this chapter we explore API workarounds and the knowledge that they contain.

6.3 API Workarounds: A Qualitative Study

We present a qualitative study on Stack Overflow posts where API users inquire about workarounds for APIs. More specifically, we would like to uncover reasons why developers request API workarounds and their solutions.

6.3.1 Collecting API workaround related posts

As presented in Section 6.2, we know that there exist Stack Overflow posts that inquire about API workarounds. Example posts, like the one presented in Figure 19, show that API users can request workarounds to request extra functionality. Prior work has shown that API users can use workarounds to bypass API defects [245]. There may exist multiple reasons for API users to seek API workarounds. However, the reasons why API users request workarounds have not been clearly established. We therefore seek to systematically determine the various reasons why API users request API workarounds to understand how they could be prevented. Furthermore, we also seek to determine how API workaround requests are answered to determine whether all API workaround requests actually require API workarounds as answers. To determine why API workarounds are requested, and how these requests are answered, we conduct a search on a Stack Overflow post dataset that was released on 5 Jun 2018 [251]. The dataset consists of over 40M Stack Overflow questions and answers.

There are over 252 thousand posts in the Stack Overflow dataset that contain the keywords "API", "library", "framework", or "interface". Therefore, it is not feasible to manually search all Stack Overflow posts to discover API workaround posts, some automation must be used to simplify the task. An n-gram classification approach was chosen for its comprehensibility and high classification rate [41]. We found that unigrams and bigrams that contained the words "workaround" or "hack" in

our dataset were very rigid and produced limited results when compared to trigrams. Using unigrams and bigrams also provided too many posts unrelated to the topic at hand. For example, the word "hack" appears in many contexts unrelated to API workarounds; this provides many false positives without the context provided by trigrams. Finally, we settled on using trigrams after first attempting to use unigrams and bigrams unsuccessfully.

To obtain a manageable number of posts for manual examination, we followed the process outlined below:

Step 1: Preprocessing. Using the open source Python natural language toolkit (NLTK) [202], we removed all punctuation and xml markup and made all strings lowercase. We further preprocessed the data by removing all stop-words (e.g., and, or, the) using NLTKs predefined list of English stop-words.

Step 2: Topic filter. We filtered out all posts that did not contain any one of the "api", "library", "framework", or "interface" keywords.

Step 3: Trigram frequency. Using NLTK, we built a dataset of all trigrams found in our topic-filtered dataset and ordered them by frequency of occurrence.

Step 4: Selection of relevant trigrams. Based on our list of frequent trigrams we manually selected trigrams that contained a logical leap to posts relevant to API workarounds, and that had a frequency higher than one and contained the keywords: "workaround" or "hack".

Step 5: Filter posts by trigram. Finally, we collected all posts in the topic filtered list (obtained in Step 2) that contained instances of the trigrams selected in Step 4. We manually selected 11 trigrams in Step 5, for example: "workaround, could, use". These 11 trigrams were frequent and only accepted if they were logically sound to all of the authors [1].

We obtained 1,846 posts by using the filtration steps outlined above. The score of a Stack-Overflow post is meant to be a marker of popularity and hence an indirect indicator of value to Stack Overflow users, we chose to rank the posts by score. Finally, we selected the top 400 scoring posts as a subset to use for our manual study. We chose to use top scoring posts, instead of randomly selecting posts, since high scoring posts are the most likely to have an impact on users, and therefore give us insight on the types of API workaround questions and answers that users consider valuable. We

[1] A complete list of the trigrams used to filter posts can be found in our replication package `https://github.com/senseconcordia/API-Workarounds`.

consider question and answer pairs as API workaround inquiries.

6.3.2 Qualitative analysis of posts

Our goal is not to find the root causes of each of our selected Stack Overflow posts. Instead, we aim to understand developers' motivations when asking for API workarounds. Furthermore, we also seek to understand what kind of answers are commonly accepted by API users. Therefore, for each workaround-related post, we examine the title, question post, accepted answer or highest rated answer, as well as any comments related to the question or answer. Investigating an API workaround post is a non-trivial task, since each post requires the investigator to understand the context, considerations, and concerns of the API users.

To reach a generalizable understanding of API workarounds, we followed a systematic process to analyze each question and answer in our dataset. We chose to use open card sorting, a commonly used sorting practice [153] that allows the sorting of posts into categories while also allowing the generation of the categories [235, 322]. More specifically, the authors of this paper performed the coding process defined below:

Step 1: Deriving base coding. A sample of 40 posts (10% of our final sample) was selected at random and given to all of the authors to code to the best of their ability. No particular constraints were set, and codes could be added at will. This step took a few days for the coders to finish.

Step 2: Discussion after code derivation. After the authors finished independently deriving their base codes, a meeting was held to discuss coding conflicts and reach consensus on a base coding that could be used for the rest of the sample. The meeting took one to two hours.

Step 3: Refining post coding. Each author independently coded another 40 posts, after which we held another meeting to discuss disagreements and refine any coding misunderstandings. Coding the posts took a few days for the coders to finish and the refinement meeting took about an hour.

Step 3: Complete coding of posts. Using our refined coding, each author independently coded the final 320 posts and revisited their prior coding. We measured our inter-coder agreement (see Section 6.3.3) after this step.

Step 5: Resolve disagreements. We discussed every conflict in the coding results until a consensus was reached for each disagreement. Conflicts were resolved by

revisiting each issue together and discussing the reasoning behind each author's coding until a consensus could be reached. Conflicts were resolved in three one-hour conflict resolution meetings.

To encourage the replication of our results and allow further studies related to API workarounds, we have made our compiled Stack Overflow API workaround questions and answers data publicly available as part of a replication package.

6.3.3 Measuring coder agreement in our qualitative study

To ensure that the coding derived in Section 6.3.2 is reliable we must have a quantitative evaluation of reliability, we chose to use Inter-coder agreement, a metric that can be used as a measure of reliability for coding results [13,132]. Coder agreement is important for trust and reproducibility. Low agreement may lead to non-reproducible results.

We used Krippendorff's α [13, 131, 132] to measure the inter-coder agreement of our qualitative study since it is a general and standard reliability measure [97].

Krippendorff's α requires a single value to be assigned to each coded item [132]. Since our coding schema allows posted answers to be simultaneously coded into multiple categories, we must modify the way we calculate Krippendorff's α slightly. We consider each category for each coded item as a separate coding unit. Coder agreement is then considered on a per-unit basis, which allows us to consider posts that have multiple coded categories. [2]

6.3.4 Qualitative study results

Table 23: Qualitative study reliability coefficient (Krippendorff's α)

	Krippendorff's α
Question categories coding	0.848
Answer categories coding	0.810

A Krippendorff's $\alpha \geq 0.800$ demonstrates reliable agreement [132]. As shown in Table 23 the Krippendorff's α for our question coding is 0.848, and the Krippendorff's

[2]The total frequencies for categorized answers exceeds the total number of posts since posts can be placed into multiple categories (e.g. *Not supported/Use another API*).

α for our answer coding is 0.810. Therefore, based on Krippendorff's α **the results of our qualitative study are reliable**.

The coding and categorization process described in Section 6.3.2, allowed us to determine four general API workaround question types and two general API workaround answer types used by API users on Stack Overflow.

Why do API users ask for API workarounds?

Through our manual evaluation of Stack Overflow posts we uncover and categorize reasons why API developers request API workarounds. Three of the four general API workaround question types contain more specific types. All of the question categories are detailed below and examples for each category can be found in Table 24.

Question Type	Quote	Frequency
Help with API dependencies	"I'm wanting to use the python-amazon-product-api wrapper to access the Amazon API [...] Unfortunately it relies on lxml which is not supported on Google Appengine."	**10**
Missing desired functionality		**150**
Data/information	"In .NET Framework, we can use [...] to get the system directory [...], but that property does not exist in the current versions of .NET Standard or .NET Core. Is there a way to get that folder in a .NET Standard"	21
Feature	"Is there an equivalent to getLineNumber() for Streams in Java 8? I want to search for a word in a textfile and return the line number as Integer."	121
Interface	"Is this somehow not what the API is meant for? Anyone know a workaround, or some kind of extra parameter(s) I could send to make it work?"	8
Requesting an improvement to the API		**28**
Functional	"Is there a better way to do this? I wish I could add my mock instances to the Laravel IoC container and let it create the commands to test with everything properly set. I'm afraid my unit tests will break easily with newer Laravel versions"	19
Non-Functional	"But I'm curious if anyone else knows of [a more] efficient way to to a bulk insert using EF Code First?"	9
Runtime problems while using the API		**159**
Backwards incompatibility	"All this works great in MVC3 (test again today, it really works) but it seems that the ExecuteCore in BaseController is not fired any more in MVC 4 beta."	20
Unexpected behavior	"Previously, I have a set of Google Drive API code, which works fine in the following scenarios [...] Few days ago, I encounter scenario 2 no longer work [...], whereas other scenarios still work without problem."	139
Unusable		**53**
Useless (Unrelated to API workarounds)		53

Table 24: Categories of questions derived from Stack Overflow posts on API workarounds

Help with API dependencies. Users sometimes seek help with API dependencies,

for example when two APIs have dependency conflicts, users might ask if a replacement exists, or if there is a way to work around the conflict. These inquiries can involve multiple libraries and build systems. This category, while infrequent could be used by API developers to determine potential compatibility problems.

Missing desired functionality. API users can have broad expectations for APIs. Users sometimes expect APIs to provide functionality that they believe should be provided by the API or that they have seen in other APIs. API users request three general types of functionality, access to extra data or information (Missing Data/information), new or missing features (Missing Feature), and they sometimes also request another interface to deal with more or fewer parameters (Missing Interface). Missing desired features are the second most common question category in our dataset. Furthermore, they appear to present common answer patterns when they require the implementation of a workaround.

Requesting an improvement to the API. API users do not always request new or missing functionality. There are cases where users are aware of existing functionality but desire some improvements. In some cases, the improvement is functional, like when a user requests an extension point for existing functionality. In other cases, the desired improvement is non-functional, like when a user requests better performance or improved security.

Runtime problems while using the API. Runtime problems present the most common API workaround question category. However, most runtime problems express themselves as general unexpected behavior. Unexpected behavior is a broad category that spans from defects to ambiguous documentation. Unexpected behavior questions present themselves when a user experiences behavior that is unexpected to them and asks for a workaround to avoid the APIs unexpected behavior. Any behavior that is unexpected by the API user falls into this category, therefore it should be no surprise that 10/13 "User is confused" answers are in response to unexpected behavior questions. Some runtime problems are more specific and can be narrowed down to backwards incompatibility. API version issues and the migration between API versions is a well-known problem that has been studied extensively [48, 128, 135, 198, 209, 260].

API workarounds contain valuable knowledge for API developers. The workarounds indicate API users' needs, such as adding features, accessing information, and bypassing runtime problems.

How are API workaround inquiries answered?

Through our manual evaluation of Stack Overflow posts we also uncover and categorize how API developers answer API workaround inquiries. We observed three main categories of answers to API workaround questions. These three main categories can be further divided into a total of eight API workaround answer categories. The answer categories were manually determined as shown in Section 6.3.2 using the post selected as an answer by the original poster. If no answer had been selected by the questions author, we selected the highest scoring answer as the best answer. Furthermore, the total frequency of answers is greater than 400 since answer posts can fit into multiple answer categories.

Already supported by the API. 28.9% (111/384) of the useful answers we extracted suggested that the posted API workaround inquiry was already supported by the API in some way. In most cases the user had to make a small adjustment to their implementation. In 24 cases, the API could be used "as is" and addresses the inquiry without any modification. Finally, in nine cases, the inquiry could be answered by using the API, but the user had to change their current implementation to fit the API requirements.

Not currently supported by the API. In most cases, API user inquiries present a need that cannot currently be addressed with support from the API. In such cases, the API users will have to produce some extra code to implement a workaround. Suggested workarounds vary in scope but follow general patterns to add features, access information, and work around runtime problems. In 80 cases, accepted answer posts suggest using another API to address the API user inquiry. In some cases, a solution to the inquiry is available or will be available soon, but only in the form of an update or patch. In 38 cases the inquiry is simply not supported by the API by design. Finally, in three cases an answer could be provided, however the posted answer did not recommend using a workaround.

User is confused. In 13 cases we encountered answers that suggested that the

Answer Type	Quote	Frequency
Already supported by the API		**111**
Change your current implementation	"I have found the solution. I switched the direction of this mapping"	9
Use API as is	"As others have said, there's nothing wrong with using [...]"	24
You will need small adjustments	"The simplest way is to just append [...] to the end of your command"	78
Not currently supported by the API		**260**
Need to implement a workaround	"One possible workaround is to write "setter/getter-like" methods, that uses a singleton to save the variables [...] or — of course — write a custom class [...]"	107
Not supported/No solution	"The reason you can't do this is because it is specifically forbidden in the C♯ language specification"	38
Not recommended	"This is asserted as "by design" [...]. Consider a post-processing step that hacks the paths the way you want them."	3
Use another API	"Do you absolutely have to use java.util.Date? I would thoroughly recommend that you use Joda Time or the java.time package from Java 8 instead."	80
Wait for/apply new version/patch	"I think you are experiencing a likely symptom of [...]. This bug exists in 3.2 and higher and was only fixed recently (4.2)."	32
User is confused	"Don't hack something together using JavaScript, as soon as Twitter makes an update to their widget, that's it, you're screwed. Use a server-side language and do it properly as per their documentation."	**13**
Unusable		**73**
No answer		20
Useless (Unrelated to API workarounds)		53

Table 25: Categories of answers derived from Stack Overflow posts on API workarounds

user was either misusing an API or following bad practices that were hindering their progress. As previously mentioned, most (10) of these users believed that they were experiencing unexpected behavior, when in fact the behavior should have been expected given their misuse of an API.

Unusable inquiries

Due to the nature of the heuristics we used to filter the Stack Overflow post dataset [251], we expected some false positives to make it through. Therefore, as part of our coding process we also categorized any posts we deemed to be unrelated to API workarounds as useless. 53 posts out of 400 were ultimately identified as unrelated to API workarounds. Many of these posts were either asking for opinions on APIs or focused on tools. Although these inquiries can be useful to the community, we

ultimately determined that they did not provide additional knowledge to help understand why API users seek workarounds, or the kinds of workarounds they use. Furthermore, since we separated the coding of questions from their answers this allowed us to consider the knowledge imparted by a question even if no answer existed at the time of our study (20 cases).

Addressing the needs of API users often requires producing some extra code to implement a workaround. The suggested workarounds vary in scope but follow general patterns.

6.4 Patterns for Implementing API Workarounds

Section 6.3 shows that a considerable number of API workaround requests require extra implementation from the API users. Therefore, we would like to identify workaround implementation patterns to show API developers how their APIs are used in unexpected ways. These patterns can then be used to inform the future API development decisions of API developers.

We read every post in the "need to implement a workaround" API answer category from Table 25 and found three generalized API workaround patterns. The three patterns were manually determined by the authors from recurring similarities in workaround answers. Similar questions were amalgamated into the general patterns found in this section. More patterns could likely be extracted from the data; however, our goal was not to extract every possible pattern, but to conduct an exploratory study of likely API workarounds. Therefore, we present three patterns that were manually developed based on real examples of API workaround requests by the authors of this paper. These three patterns are not an exhaustive list of patterns that could be derived from our dataset. For each workaround pattern we provide a description of the pattern, the motivation of the API users that implement such a workaround, and more importantly, the benefit of knowing such patterns for API developers. In addition, we present a code example of each pattern (presented in Figures 21, 22, & 23).

Workaround Pattern 1: Functionality extension

```
public static class PortofinoGetGeneratedKeysDelegate extends GetGeneratedKeysDelegate implements
        InsertGeneratedIdentifierDelegate {
  ...
  //Override in order to unquote the primary key column name, else it breaks (at least on Postgres)
  @Override
  public Serializable execNExrct(PreparedStatement i, SessionImplementor s) throws SQLException {
    s.getTransactionCoordinator().getJdbcCoordinator().getResultSetReturn().executeUpdate(i);
    ResultSet rs = null;
    try {
      rs = i.getGeneratedKeys();
      return IdentifierGeneratorHelper.getGeneratedIdentity(rs,
        unquotedIdentifier(persister.getRootTableKeyColumnNames()[0]),
        persister.getIdentifierType());
    } finally {
      if (rs != null) {
        s.getTransactionCoordinator().getJdbcCoordinator().release(rs, i);
  }}}
  protected String unquotedIdentifier(String identifier) { //This is a hack.
    if(identifier.startsWith(dialect.openQuote() + "")) {
      return identifier.substring(1, identifier.length() - 1);
    }
    return identifier;
}}
```

Figure 21: Functionality extension API workaround pattern example

Description: This pattern presents itself when API users extends the existing behavior of an API to add functionality that does not currently exist as part of the API, or to modify existing behavior to work as they desire.

Example: The example in Figure 21 shows an extension of the Hibernate ORM API to support functionality for Postgres databases, that require an unquoted primary key column name. Standard behavior is therefore modified to unquote the identifier to work around the different behavior required by Postgres.

Motivation: This pattern appears when users desire an unavailable functionality from an API. This workaround pattern allows API users to circumvent existing functionality without removing or breaking any of the existing functionality. This allows API users to keep all existing API functionality and have their desired workaround included as well.

Detection strategy: To detect this pattern we attempt to determine the frequency

of API class extensions, as well as the frequency of method overrides. We can compare this data to a baseline of non-extended API class invocations, and non-override API method invocations. Abstract classes should be ignored since they are designed to allow flexibility for the user to create whatever they want

The intuition behind this pattern is that if a class or method is more often extended or overridden than it is invoked, then the functionality of the class or method is not offered in the way most often desired by the API users. Therefore, the data for this pattern should present cases of functionality improvements for API developers. Clone detection approaches can also be used to check if common functionality can be found between projects.

Benefit to API developers: Instances of this pattern can present API developers with real scenarios for desirable features, and hints to implement them, without direct communication with users. Therefore, API developers can use instances of this pattern to determine what desirable functionality is missing from their API.

Workaround Pattern 2: Deep copy

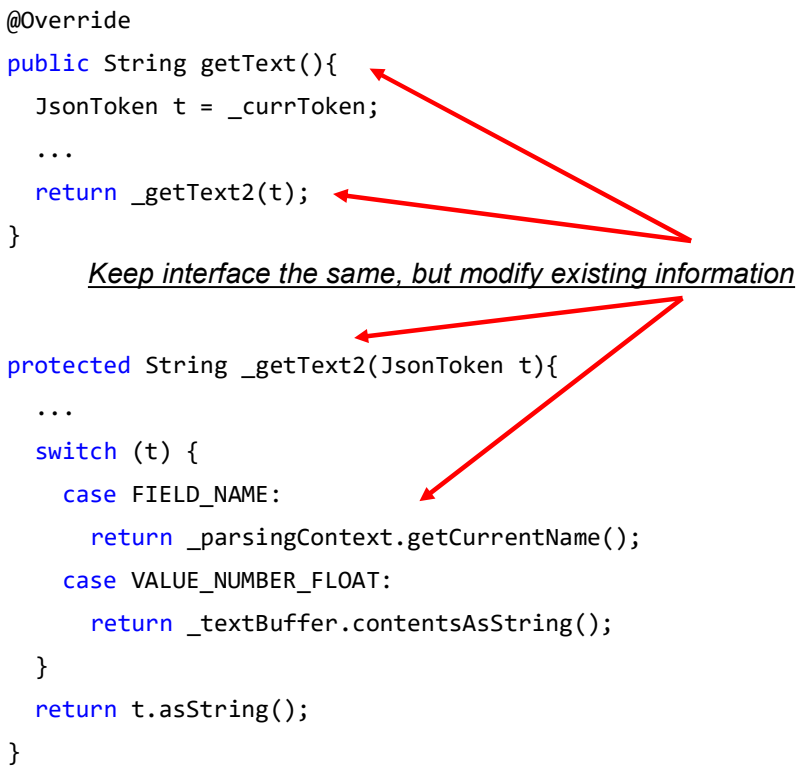


Figure 22: Deep copy API workaround pattern example

Description: This pattern presents itself when a user attempts to copy API data to use a copy locally rather than directly use the API functionality. This can be done to add or modify functionality or to work around a software defect.

Example: The Jackson API includes parsers for several data formats (ex. Avro, CSV, XML, YAML). However, it does not contain a parser for BSON or Rison, therefore users must create their own parser to support these data formats by providing a new interface that copies existing functionality but provides Rison or BSON compatible outputs. The example presented in Figure 22 shows a method that access existing information in the API modifies it and provides the information through the usual API.

Motivation: This pattern specifically looks at cases where a user wants to use the data provided by the API rather than the methods provided by the API. In this pattern, API users extract internal API information to add or modify API functionality in their application. This allows the users to maintain complete control of the functionality in their application while relying only on the API's data.

Detection strategy: To detect this pattern in Java applications we can look at the API fields and API getter method usage in GitHub projects. We believe that looking at fields and getters that are often called by API users can give insight into the usage patterns of these API users. This insight coupled with an understanding of the API architecture can explain where new interfaces could be created. We also look at classes that use a high number of distinct fields and getters to determine how and why users are using the API data.

Benefit to API developers: This pattern tells API developers that their API contains desirable data, but that functionality to use this data is missing or defective. Therefore, interfaces should be modified or added to provide desired functionality.

Workaround Pattern 3: Multi-version

Description: This pattern manifests when API users attempt to use two or more versions of an API to work around a runtime problem (e.g., bug) or introduce functionality found in separate API versions.

Example: The Log4j and Log4j2 APIs allow the user to set logger context, however some early versions of the Log4j2 API experienced some issues with exception logging. By using a classLoader and a JAR of Log4j it was possible to dynamically load and

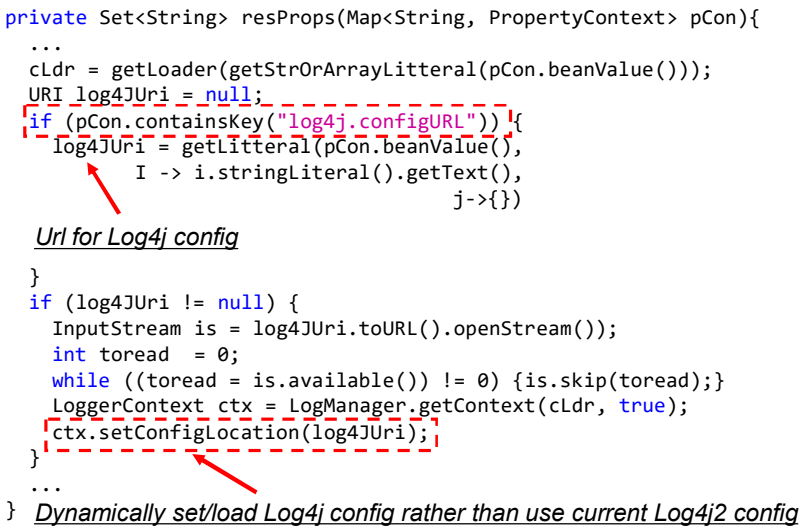


Figure 23: Multi-version API workaround pattern example

use the log4j logger context to circumvent exception logging issues experienced by Log4j2. This is presented in the example[3] in Figure 23.

Motivation: Many workarounds are requested to deal with defects in APIs, we found a wide range of solutions for this problem in our Stack Overflow dataset. However, we found some cases where users are encouraged to use an older or newer version of the API to resolve an issue (i.e. bug).

Detection strategy: To detect this pattern we can attempt to determine when users use two or more versions of an API in a given project. In the case of Java, it is not normally possible to statically load two versions of the same library because the class paths would conflict. However, it is possible to dynamically load two (or more) JAR files using class loaders at runtime and then use the functionality from any or all of the loaded JARs as desired. We can therefore attempt to determine instances of this pattern by detecting when a given class or method shows support for more than one version of an API.

By storing various versions of each library (in JAR format), and by looking at all API method invocations for each project we can determine multiple version usage. Most APIs keep functionality the same across multiple versions, however some APIs will change. Therefore, if we detect that a method invocation maps to a specific API version, but the rest of the project maps to different API versions, we can flag this

[3]This problem has since been resolved in Log4j2, however one of the user projects we encountered still maintains a workaround for this issue for unknown reasons.

API method invocation as suspicious.

Benefit to API developers: Using a different version of an API is also sometimes suggested as a solution for missing desired functionality that existed in an older version of an API. Therefore, API developers can use instances of this pattern to detect potential defects (and their solutions), as well as desirable functionality, directly from user projects.

6.5 Reporting API Workarounds to Developers

In this section we present a study conducted to detect the existence of API workaround patterns in real-life projects that use the APIs. Furthermore, we discuss the results of our study and the API developer responses to the reported patterns.

6.5.1 Identifying API workarounds in real-life projects

Since our generalized API workaround patterns are based on data collected from Stack Overflow, we do not have direct evidence that these patterns can readily be found in real-life software projects. Therefore, we produce an experiment to confirm the existence of these patterns in open source projects.

Our detection strategies rely on parsing API source code and extracting binding information for fields, methods and classes in the API with the help of the Java abstract syntax tree parser [117] and symbolic link resolver JavaParser [117]. Once an API has been parsed, any number of user projects can be targeted to detect the occurrence of workaround patterns inside those projects. If a workaround pattern is detected, we manually observe the identified candidate and report the candidate as a possible improvement to API developers.

Subject APIs

We selected five open source APIs, all of which have their source code available on GitHub and compiled JARs available in Maven repositories. We selected APIs programmed in Java to limit the scope of our experiments. However, it should be noted that our generalized patterns are language agnostic and were generalized from Stack Overflow posts without filtering by programming language. All of the chosen

APIs are popular open source APIs that have been used by hundreds of public GitHub projects. The popularity of the APIs allows us to obtain varied uses of the APIs.

Guava *(394 user projects)* Google's Guava API is an open source set of commonly used Java libraries. The API includes APIs for concurrency, primitives, hashing, and many other functionalities [94]. Prior research on 10,000 GitHub projects has shown that Google Guava was the 8th most popular Java library in 2013 [285]. We targeted 20 different versions of the Guava API, from version 20.0 to version 27.1.

Hibernate *(642 user projects)* Hibernate is a free and open source framework that provides mapping from Java classes to database tables as well as abstracted data querying [222]. We targeted all 12 of the minor releases of the Hibernate API that were available on the maven central repository, from version 3.3.2 to version 5.4.2.

Jackson *(588 user projects)* The Jackson Core is an open source Java API that provides a JSON parser/generator with other data encodings, such as CSV, XML, YAML and more [80]. We targeted all 10 of the minor releases of the Jackson API that were available on the maven central repository, from version 2.0.6 to version 2.9.9.

JUnit *(1,000 user projects)* JUnit is an open source unit testing framework for Java [26]. Prior research on 10,000 GitHub projects has shown that JUnit was the most popular Java library in 2013 [285]. We targeted 20 different versions of the JUnit API, from version 3.7 to version 4.12.

Log4j *(475 user projects)* Apache Log4j is an open source Java logging framework [83]. As of the writing of this paper, over 4,260 maven artifacts have the Apache Log4j Core as a direct dependency [82]. We targeted 20 different versions of the Log4j API, from version 2.0.1 to 2.11.2.

API user projects

The API user projects chosen for this paper were all open source projects hosted on GitHub and selected based on their use of the five Java APIs we selected. We first searched all of GitHub for README files that mention the name of our target APIs. There is no current tagging system on GitHub to search for APIs used by GitHub projects. Therefore, we rely on heuristics to determine if a project uses one of our five target APIs. We found that if a project README mentions an API by name,

it is likely that the project will in turn use this API. Furthermore, we used project "stars" as a metric for popularity of a project. Although GitHub "stars" are not an indication of quality, it is an indirect measure of popularity. If a project is more popular, it is possible that it will have a larger impact, and the information obtained from examining this project should therefore be more important to API developers. The minimum project size was set at 5MB in order to reduce the number of dummy projects that might contain no code. Using these filtration criteria, we either selected all of the projects that met our criteria or the top 1,000 starred projects for each target API, whichever came first. The number of projects used for each of our selected APIs can be found in Section 6.5.1.

Each of these projects was used to attempt to map binding information obtained from the API source code to API uses in the user projects. Furthermore, we also used the JARs for each API to determine version specific binding information. We applied our pattern detection strategies to determine if one or more of our three patterns are present in a user project.

Detecting patterns

To help detect the API workaround patterns presented in Section 6.4, we produced scripts and prototype tools based on the detection strategies presented in Section 6.4. The prototype tools and scripts are publicly available [4].

API developers are most likely to be interested in active API workarounds, we concentrate on the latest releases of API user projects. By using the latest releases of user projects we can keep our results relevant to API developers, circumvent a number of build problems related to older versions [274], and reduce the pattern detection time. Furthermore, since JavaParser [117] does not require building projects to obtain an AST or to build symbolic links, we can parse API user projects without the need to worry about build issues [274]. As a first detection step, we leverage JavaParser [117] to extract API class names, API method declarations, API field declarations, and specific API methods that contain the keyword 'get'. This information can later be used to map API declarations to API user invocations by further leveraging JavaParser [117] to obtain code bindings in user projects.

[4] `https://github.com/senseconcordia/API-Workarounds`

Functionality extension: To detect the *Functionality extension* pattern, we use the binding information obtained through JavaParser [117] to extract all API method overrides, API method invocations, API class extensions, and API class invocations for all of the API user projects in our sample. We then build a frequency map of all of these, to determine which classes are more often extended rather than invoked and which methods are most often overridden rather than invoked. Based on this data, we observe the items with the highest extend:invoke and override:invoke ratios.

Deep copy: To detect the *Deep copy* pattern we use JavaParser [117] to extract API field invocations and API getter method invocations from API user projects. We keep track of how many of these items are invoked in a given API user class, and the global invocation patterns across all API user applications. We then consider API user classes that use the most API field and API getter invocations as potential *Deep copy* candidates.

Multi-version: We conduct heuristic analysis on the JAR releases of our target APIs to determine links between target APIs and API user applications. By using these links, we can heuristically determine which API versions are compatible with a given user application. Through the heuristically determined links between API JARs and API user applications, we can determine which user applications would require more than one version of an API to support all of their API calls. Using this information we can flag API user applications that require multiple versions of an API.

Using our detection strategies, we produced lists of API user code instances that were likely to contain API workaround patterns. We manually verified the top 10 most likely workaround candidates for each API for each pattern, giving us a total of 150 manually verified potential API workaround pattern instances. Any candidate deemed an API workaround instance, after manual verification, was reformulated by the authors as an API feature request and sent to API developers, either through GitHub or their forums. We detect API workaround patterns in API user applications of varied maturity, without knowing which version of the API is used a priory. Therefore, we do not originally know if any of the workarounds we find in user applications have since been used as actual improvements and bug fixes in more recent versions of the API. If we detect a workaround in an old user project and later determine that the workaround has been integrated into the API, we therefore consider

this an indication that API developers could benefit from knowledge of API user workarounds.

6.5.2 Results and discussion

We breakdown our manual observations of 150 potential API workarounds that were detected using automated approaches. We first manually examine all the 150 instances to confirm whether they indeed correspond to workarounds. We find that 80 out of the 150 (53%) instances are true instances of workarounds. We manually check the reasons of the instances that are detected by our patterns but not workarounds. We find that there is a single non-workaround instance of *Functionality extension* pattern. The pattern instance was a custom extension of a Hibernate exception and therefore considered normal usage of the API. We find 20 non-workaround *Deep copy* pattern instances that were exclusively for Jackson and JUnit. In those instances, the top fields and getters copied by API users to ease their testing code, i.e., JUnit. Finally, all of the detected instances of *Multi-version* patterns belong to JUnit, Guava, and Log4j, since the Hibernate and Jackson APIs did not present any instances of the pattern. However, the majority (49 out of 50) of our *Multi-version* pattern instances appear to be defensive coding rather than pure workarounds.

To avoid requesting too much information from the developers of the studied subjects, we strategically pick manually verified true instances to submit feature requests. In particular, we concentrate on more complex functionality addition or modification, and defect workarounds that could clearly be discerned by the authors.

Functionality extension:

During our manual observation we extract nine *Functionality extension* API workarounds from the selected user systems. By searching through forums and patch notes, we find that three of the *Functionality extensions* did not exist in the APIs when the user projects created workarounds, but they had already been incorporated into the APIs when we searched their forums. This confirms that our patterns are indeed detecting functionality that was missing from the APIs.

In two cases, we find currently existing pull requests that are in the process of being integrated into the APIs. For example, pull requests are in discussion for

SQLite support in Hibernate and users have posted that they *"Would love to see official SQLite support in Hibernate"*. Therefore, in five cases, desired functionality had been deemed valuable by API developers and was either integrated into the APIs or is currently in the process of being added.

In two cases, we found two existing but unfulfilled feature request posts in the API forums or on Stack Overflow. In one case, a Stack Overflow post (post id: 2308543) details the unexpected behavior and the desire for this feature. This shows a real user desire for this feature. However, the feature has still not been added.

In two cases we created feature requests for missing functionality. As of the writing of this paper, one feature request is in the APIs feature request queue. The other feature has been acknowledged by the API developers as desirable by users, but they do not have the resources to maintain that functionality at this time.

Deep copy:

We extract two interesting *Deep copy* API workarounds from our dataset. We created feature requests for new functionality to improve the APIs. We received positive responses to the functionality that was proposed. When we requested a BSON format addition, one API developer replied that they did not want to support the functionality, but that *"[...] BSON-backed streaming api implementation makes sense (dataformat module) – this is what is used to support dozen other formats."*. Furthermore, they pointed us in the direction of an existing third-party package that could supply this functionality. This shows that the functionality was indeed desirable enough for someone to create a third-party library for it. This therefore confirms that the detected workaround pattern contained functionality that was not provided by the API. Furthermore, the third-party library with this functionality shows the value of our reported workaround functionality.

Multi-version:

We find 50 cases where multiple versions of APIs were used. However, only one case was providing a workaround for missing functionality. We examined this case in detail and found that an issue did exist with the API. However, the issue had been fixed by

a patch soon after the issue was introduced. The fix shows that workarounds would potentially assist developers in identifying problems with their APIs used in real-life by API users.

In the 49 other cases, after careful examination of the API user code and documentation, we determine that API users sometimes code defensively to allow their users to use a wide range of compatible libraries. In those cases, API users will have a direct dependency on a specific version on an API, which they will bundle with their project. However, they will in turn allow their users to use a range of different API versions, which will be dynamically loaded and override default behavior to provide compatibility with newer API versions. If API developers had knowledge on which API combinations users most often employ, this could direct their testing efforts to maintain compatibility between API versions.

Unnecessary workarounds:

User code can sometimes present an API workaround pattern with code that simply emulates existing functionality. We found three instances of user code that presented as workaround patterns but could have been implemented using existing API functionality. In these cases, perhaps a lack of understanding of the APIs functionality by the API users is at fault. This could be mitigated by improving API documentation and examples. API developers can use this information to efficiently spend time on APIs that have documentation issues and generate examples specifically for those APIs.

Based on the responses to API workaround feature requests, both already existing and those we created, we can confirm that API workaround patterns detected in API user projects can provide valuable knowledge to API developers[5]. Furthermore, we can confirm that the three patterns presented in Section 6.4 of this paper do exist in API user projects, and that they are used to provide missing functionality and work around unexpected behavior.

[5]A list of detected patterns and feature requests is available in our replication package.

6.6 Threats to Validity

Construct validity. We do not claim to have found all inquiries pertaining to API workarounds. However, we believe that the sample collected is adequate to produce an exploratory study into the problem at hand. Although we diligently attempted to confirm the detected instances of implementation patterns for API workarounds by searching application documentation and online forums, and we reported issues to API developers, it is still possible that the patterns detected in user applications were misidentified as API workarounds. We do not claim to be experts for any of the user applications studied nor for any of the APIs selected. We do not claim to have found or reported all existing workarounds in the studied systems. However, investigations into the instances detected appears to confirm the existence and usefulness of the patterns. Future empirical and user studies can be done to complement our study and may bring additional insight to our results.

External validity. Since the API workaround pattern instances in this study were detected in Java APIs, it is possible that the findings in this chapter do not generalize to other programming languages. However, while the strategies presented in this chapter were tested on five Java APIs, the strategies were developed based on language agnostic Stack Overflow posts and should therefore apply to a range of programming languages (e.g., C#).

Internal validity. The API workaround inquiry categories and patterns presented in this chapter might not be fully indicative of API workarounds and instead reflect internal bias. We attempted to mitigate these threats by having the reviewers independently label and reach a consensus on the categorization of Stack Overflow posts and the implementation patterns extracted from these posts. Our manual observation of 400 Stack Overflow posts may also present internal bias, future studies involving more posts can complement our results. However, we reported API workaround patterns to API developers and received feedback that suggests that the workarounds we detected are actual workarounds and should be considered valuable for future fixes or extensions to the APIs.

6.7 Chapter Summary

We conducted an exploratory study on API workarounds. By studying inquiries from Stack Overflow, we find that API users seek API workarounds to add desired functionality, improve APIs, and resolve unexpected API behavior. Furthermore, we show that many API workarounds require extra code from API users to implement workarounds. Using workaround implementations suggested in Stack Overflow answers, we extract three generalized API workaround patterns that are implemented by API users to deal with missing API features and unexpected API behavior. We find real-life examples of these patterns in open source projects and report instances of these patterns to API developers. Without our findings, these patterns might be misidentified as general development and developers might ignore their unique characteristics. We find that API developers consider these workaround instances as real issues, and either add them to their issue tracker, or encourage pull requests to remedy them. This chapter makes the following contributions:

1. We are the first to study inquiries that concern API workarounds.

2. We introduce and confirm the existence of three general implementation patterns for API workarounds.

3. We determine the usefulness of these patterns to practitioners through their adoption into API code bases.

Our findings highlight the benefits of using open-source repositories to uncover API workaround usage patterns. We show that these workaround patterns can be used to reduce the knowledge gaps between API developers and API users by allowing API developers to improve their APIs through the knowledge they acquire from these repositories.

Part IV

Conclusions and Future Work

Chapter 7

Contributions and Future Research

Software application programming interfaces (APIs) are now an imperative part of Software Engineering. For their users, the usage of APIs can speed up development and reduce project overhead. For their developers, APIs present business opportunities.

The evolution of these APIs requires constant effort from their developers and users alike. Through their very nature, APIs create a separation between their users and developers. This separation results in knowledge gaps on both fronts. In this thesis, we empirically studied knowledge gaps incurred by API users and developers and precipitated by the very APIs they use and develop. In this section, we outline the contributions of this thesis and present future research topics.

7.1 Thesis Contributions

The underlying goal of this thesis is to assist with API evolution from the perspective of both API users and API developers and to propose solutions to challenges incurred by this evolution. We leverage the knowledge contained within open-source software repositories. Through our literature review in Chapter 2, we first identify issues of API evolution that would particularly benefit from knowledge that could be contained within open-source software repositories. We then investigate three API evolution issues (i.e., Android API migration, API misuses in Java, and API workarounds) in four empirical studies. We highlight the primary contributions of these four studies below:

1. **What are the Challenges Associated with Android API Migration?** We find that although a portion of the removed or deprecated Android API methods do not have a replacement, identifying a replacement using documentation or historical code change information is not a challenging task for practitioners. Existing tools could mine this information without much issue. However, identifying migration replacements are not the only challenge for Android API users attempting API version migration, we also identify other challenges, that are more time consuming to address. We find that: not all modified methods have migration pathways, potentially forcing Android API users to continue using old APIs or resort to workarounds; historical code data is often a sufficient source of information when documentation is lacking; Android API migrations are sometimes introduced by Android API developers long before an API method is removed or deprecated, this has implications for heuristics used to search for replacement APIs; most Android API users only require a few common API methods, these methods should get the majority of the migration support. Finally, we find that Android API migrations often require code modification that require developer knowledge. This knowledge must in turn often be obtained from sources other than documentation such as the Android API source code repository or existing API migration solutions hidden across multiple open-source repositories. Our approach A3 is a solution to this problem.

2. **Using Existing API User Knowledge as Android API Migration Aid:** We propose an approach (A3) to acquire developer knowledge that was identified in our empirical study as necessary for many Android API migrations. This knowledge is contained within existing Android API migrations conducted within existing Android API user projects. Our approach assists developers with Android API migrations by learning Android API migration patterns from code examples mined directly from available code repositories. We find that it is possible to automatically extract Android API migration patterns from both public code examples and manually produced Android API examples that are created with minimal effort. By learning Android API migration patterns from these examples, our approach can provide either automatically generated Android API migrations or useful information to guide Android API users with

their migrations in 71/80 cases. Our user-study showed that the examples provided by our approach are both desirable to users and allow users to migrate Android APIs, on average, 29% faster.

3. **Improving Misuse Detection Approaches:** We manually uncover 108 cases of alternate but correct API usages that were falsely identified as API misuses by a state-of-the-art API misuse detection approach. We manually investigate these 108 cases and discover five patterns of alternate but correct API usage. We find that these patterns can be used to transform existing API usage examples into complementary artificial API usage examples. These artificial examples can in turn cover API misuse detection knowledge gaps caused by a lack of diversified existing API usage examples. We find that these complementary examples can be used in state-of-the-art API misuse detection approaches to reduce the incidence of false positive detection. Furthermore, we also find that these artificial examples can be used to complement existing examples in API misuse detection without incurring any loss of recall. Our findings highlight the potential of extracting missing API usage knowledge by generalizing API usage examples through pattern-guided source code transformations. This knowledge can in turn reduce the dependence of example-based API misuse detection on mining large and therefore costly samples of user projects.

4. **Guiding API Development by Using API Workarounds:** We are the first to establish an empirical link between API workaround requests from API users and patterns to identify instances of these workarounds in user code. We find that API users usually request API workarounds to add desired functionality, improve APIs, and resolve unexpected API behavior. Furthermore, we show that many API workarounds require extra code and some knowledge of the API for API users to implement them. We manually identify three API workaround patterns that are implemented by API users to deal with missing API features and unexpected API behavior. We confirm the existence of these API workaround patterns by finding real-life examples of these patterns in open-source projects and successfully report project specific instances of these patterns to API developers as potential avenues of improvement for their APIs.

7.2 Future Research

We believe that this thesis presents concrete contributions towards understanding and reducing the knowledge gaps between API users and API developers. However, there are still many avenues for future research. We outline some of these future research avenues below:

7.2.1 API Development Theory

Throughout this thesis we encountered a variety of APIs, not only in our data, but also when creating scripts, apps, and programs that were used for our research. By looking at this variety of APIs, it is clear that APIs vary widely in design, even within a single programming language like Java. There is currently no grounded theory that dictate what makes a "good" API. A variety of best practices can be found, but ultimately most appear to be based on developer beliefs, or organizational rules, rather than data. More systematic empirical studies should explore what engineering attributes make an API desirable to API users.

7.2.2 API Migration Patterns

While this thesis has presented an approach that mines and applies existing API migration knowledge in Chapter 4, work can still be done to improve the state-of-the-art in API migration. Particularly, common API migration patterns, such as the ones mined by A3 merit more research. For example, these patterns could identify how API users commonly chose to migrate APIs, and which APIs do users most commonly migrate. This information could in-turn be used to identify where API users need more help, and where API developers should devote more effort.

7.2.3 API Misuses to Inform API Development

Based on our findings in Chapter 5, we believe that, similarly to identifying API usage patterns that can inform API development, we could use API misuse detection tools to uncover frequent API misuses. These frequent API misuses could then be used to identify particular weak spots within APIs. Therefore, we believe future research

should understand why frequent API misuses occur and how they can be leveraged to aid API development.

7.2.4 Identifying Common API Usage Patterns to Inform API Development

Our findings in Chapter 6 indicate that it is possible to use API user development information to uncover potential areas of improvement for APIs. However, we only concentrated on a niche area of API user data, namely API workarounds. There are still many avenues to explore where and how API usage can be used to inform API development (e.g., APIs frequently used together, API migration patterns). In future work, we plan on looking at frequently used APIs to uncover general reasons why certain APIs are used together and whether these reasons can be generalized to aid API developers in developing better APIs.

Appendices

Appendix A

Java APIs Commonly Used for API Research Evaluation

APIs are evermore common in software engineering. Using APIs is now a routine part of the software development lifecycle. Research into APIs has therefore sensibly increased to match the rising adoption rate of APIs and the challenges uncovered by this growth in popularity. However, our research shows that, in research, a few common APIs are frequently used to make or test most API research inquiries. These APIs are primarily Java APIs, and may not reflect the state of API evolution as a whole. This technical report was created to highlight which APIs are most commonly used in API research, in the hopes of raising awareness of current knowledge gaps in the field to improve the status quo.

In this Appendix we present the 17 most popular systems used to evaluate software engineering research into APIs. We selected 143 published works extracted from a recent systematic review of API evolution literature and manually determined which APIs were used to either produce or test the hypotheses presented in each published work. As well as presenting the most common APIs used for research evaluation, we also highlight how each API was used to evaluate existing API research.

Java API

With 39 independent papers within our sample dataset, the most common API used to evaluate API evolution research is the Java API either through the use of its various standard libraries, or through the JDK [203]. The Java API is widespread and has a large userbase [244]. Furthermore, the Java API benefits from a large number

of open source projects available in online repositories like GitHub. For example GitHub contains 879,265 Java based projects [244]. Many papers that presents tools or approaches improve API usability [88, 151, 163, 189, 194, 221, 223, 265, 268, 287] and help with API evolution and migration [3, 11, 31, 36, 90, 127, 154, 186, 199, 204, 264, 275, 313, 318] make use of the Java API to evaluate the effectiveness of their approach. The Java API has been used to conduct empirical studies on API evolution [180, 244] and API usability [6, 59, 87, 88, 110, 124, 151, 163, 189, 194, 195, 206, 214, 221, 223, 238, 241, 265, 267, 268, 287]. The Java API has also been used to construct API quality datasets [3], and to evaluate API security tools [114].

Android API

The Android API popular for evaluating hypotheses for numerous reasons. It is a large and open source API [135], and the API benefits from a large user base through the Android ecosystem [146]. In this section, we do not distinguish between studies that make use of the Android API and Android apps to calculate the number of studies that use the Android API. We conside the Android apps presented within the context of the studies in this dataset as examples of Android API users since they are presented from the perspective of the Android API in their respective papers. A large portion of studies that use the Android API as an evaluation system conduct empirical studies on the evolution of APIs [24, 39, 77, 96, 98, 135, 146–148, 150, 162, 180, 277]. However, other studies also employ the Android ecosystem for the evaluation of various tools or approaches to help with API evolution [32, 108, 245, 300, 303], and API usability [17, 174, 210, 259, 284, 304]. Finally some papers make use of the Android API to evauate empirical studies on software usability [101, 206, 267, 283, 321], and software performance [149]

Toy systems

We consider simple systems that are produced for the sole sake of evaluating an approach presented in a paper to be toy systems. These toy systems can be used to showcase a tool. However, they are not necessarily representative of existing projects that can be found within an APIs ecosystem. 20 of the 291 publications we sampled made use of such systems. These toy systems are used for a variety of studies such as API refactoring tools [85, 100, 269, 270, 323], API documentation studies and tools [28, 43, 74, 118, 254], studies on managing API evolution [72, 73, 197, 213, 234, 311], and studies on understanding and developing better APIs [116, 169, 215, 273].

We find that 12 out of 20 studies that make use of toy systems to evaluate their findings use the Java programming language to create these toy systems. The other studies that make use of toy systems make use of varied programming languages such as BPEL [28].

Eclipse

Eclipse is an industrial yet open-source Java IDE [52]. Eclipse freely provides access to the source code to it's framework which can then be used for evaluation by researchers. Eclipse has been used as an evaluation system for API evolution empirical studies [53, 66, 125], API usability studies [38, 103, 109, 321], an API conformance checking tool [217] API evolution mining tools and approaches [158, 173, 246], empirical studies on API refactoring [64, 126], an API migration recommendation tool [52], and API refactoring detection tools [63, 262].

JHotDraw

JHotDraw is a medium sized Java GUI framework created to demonstrate design patters [297]. JHotDraw has been used to evaluate API recommendation tools [14], API usage mining tools [57], API refactoring detection tools [63, 262], API migration tools and approaches [128, 266, 295], API change rules evolution in empirical studies [66, 246, 297, 302], and refactorings in an API upgrade case study [323].

Log4j

Log4j is a Java library that provides application logging functionality [122]. Various studies that present API tooling such as, API usage extraction tools [243, 264], API recommendation tools [152], API refactoring detection tool [262], and API migration tools [122] make use of the Log4j API to test the effectiveness of their tools. However, papers that present empirical studies such as API evolution studies [48, 64, 66, 180], studies on API documentation evolution [247], and studies that observe API compatibility [115] also use the Log4j API as a benchmark.

Struts

Apache Struts is a Java MVC framework for creating Java web applications [303]. In this section we do not distinguish between Struts and Struts 2. Struts is mainly used to test API tools and approaches such as detection of refactoring in APIs [63, 262], mining framework changes [166, 246, 303], API recommendation tools [265], and tools to detect dynamic API specifications [2]. However, in two cases empirical studies use struts to validate API evolution hypotheses [48, 66].

Guava

Guava is a Java library of collection utilities that were not originally provided as part of the Java SDK [243]. Over 3,000 Guava clients exist on GitHub [243]. Guava has been used to test a variety of hypotheses, ranging from API usage analysis [242, 243] API deprecation [130, 244], API documentation analysis [267], the impact of refactoring on API clients [133], and the impact of unbundling APIs [161].

Hibernate

Hibernate is a framework for mapping an object oriented domain to a relational database [241]. Hibernate has over 1000 deprecated APIs over it's history, making it a prime candidate to test API deprecation hypotheses [241]. Hibernate has several user projects available on GitHub [210] and many questions on online forums, also making it a good candidate for approaches that learn API characteristics from online forums [53, 210], or studies that observe API usability [109, 191, 242]. It has also been used as a test subject for hypotheses about API documentation [230].

JUnit

JUnit is a popular open source Java testing framework [109]. It is used as a test subject for studies about API documentation [230, 267], API usage patterns [109, 243, 264], API evolution problems [61], and API migration [317].

JFreeChart

JFreeChart is a Java chart library with over 54 releases that contain many API changes with similar names [297]. The change history of JFreeChart makes it a good API to test change rules in APIs [297], understanding unfamiliar APIs [69], and testing API recommendation [68] and migration [128, 191, 295, 312] tools.

Proprietary systems

Not all systems used for API research are open-source systems. Six of the papers in our sample test or build their hypotheses upon proprietary closed source systems from various companies. These studies are nevertheless varied in scope, and do not appear to be limited by the closed nature of their source code. The studies range from API usability and design [179, 255, 307], extracting API usage patterns [280], and understanding API evolution [95, 310].

Spring

Spring is a framework that provides access to Java objects through reflection [243]. It is a popular project, that has at least 150 classes, and has at least 10 commits per week

over its lifetime [243]. It is employed as a test subject by six of the papers in our sample dataset. It is used a a test subject for studies in API recommendation [152], improving API documentation [53, 130], understanding developer reaction to deprecation [244], and for approaches to understand API usage [242, 243].

Hadoop

Hadoop is one of the most popular Java libraries developed under the Apache foundation [265]. Hadoop is used as a test subject by a variety of studies in our dataset. Most of the works that employ Hadoop as a test system concentrate on API documentation, either by detecting documentation errors [314], recommending or searching for API documentation [230, 267], or exploring API documentation quality [130]. However, Hadoop is also used to test API recommendation tools [265], and as a test subject to keep track of API popularity [108].

Lucene

Lucene is a free and open-source search engine Java library [263]. Six of the papers in our dataset use Lucene as an API to test their hypotheses, however none of these studies highlight why Lucene is a prime candidate as a test API. In all six instance, Lucene is selected as one of several test APIs, and Lucene never appears as a singular test API without our dataset. Lucene is used in a variety of studies, from API migration studies [187], API deprecation studies [130], API documentation evolution and error detection studies [247, 314], API specification checking studies [217], and API refactoring detection studies [262].

Pharo

Pharo is a dynamically typed programming language, with over 3600 distinct systems and over 6 years of evolution [107]. Therefore the API is seen as a good candidate for ecosystem studies. In particular studies that use the Pharo ecosystem concentrate on the ripple effects of API changes on an ecosystem [226], how developers react to API deprecation [227], and how developers react to the evolution of an API [104, 106, 107]. One of the studies in our dataset also used the Pharo API as a test subject to benchmark a tool that extracts API changes that occur during API evolution [105].

.Net API

Within our dataset, the .Net API as a test API is always coupled with the Java API. The APIs can be couple as a comparison since both APIs present similar features, and a large number of client programs [87]. However, in one case both APIs are

required to test the hypothesis since the goal of the study is to build a migration mapping between two APIs [36, 199]. In the majority of cases however, the APIs are chosen to provide results that are valid across languages, either to uncover patterns of knowledge [155], or as responses to user surveys [228].

In this Appendix we present the 17 most popular APIs used to evaluate software engineering research into APIs. We find that most of the popular APIs used for evaluation are Java APIs, with a few outliers such as the Pharo API and the .Net API. We hope that by highlighting the most common APIs used to evaluate past API research, the information presented in this Appendix can be used to foster future API research by facilitating the replication of existing work, as well as highlighting the lack of programming language variety in existing API research.

Bibliography

[1] Alberto Abelló Gamazo, Claudia Martinez, Carles Farré, Cristina Gómez, Marc Oriol, and Oscar Romero. A Data-driven approach to improve the process of data-intensive API creation and evolution. In *CAiSE-Forum-DC 2017: Proceedings of the Forum and Doctoral Consortium Papers Presented at the 29th International Conference on Advanced Information Systems Engineering*, pages 1–8, Essen, Germany, nov 2017. CAiSE.

[2] Ziyad Alsaeed and Michal Young. Extending existing inference tools to mine dynamic APIs. In *Proceedings of the 2nd International Workshop on API Usage and Evolution - WAPI '18*, pages 23–26, New York, New York, USA, 2018. ACM Press.

[3] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N Nguyen, and Mira Mezini. MUBench A Benchmark for API-Misuse Detectors. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 464–467, New York, New York, USA, may 2016. ACM Press.

[4] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. Investigating Next Steps in Static API-Misuse Detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, MSR '19, pages 265–275, Piscataway, NJ, USA, may 2019. IEEE.

[5] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Trans. Software Eng.*, 45(12):1170–1188, 2019.

[6] Davide Ancona, Francesco Dagnino, and Luca Franceschini. A formalism for specification of Java API interfaces. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops on - ISSTA '18*, volume 2, pages 24–26, New York, New York, USA, 2018. ACM Press.

[7] Android. Android developers distribution dashboard. *developer.android.com*, 2017.

[8] Android. Android developers resources, getcolor. *developer.android.com/reference*, Jul 2017.

[9] Android. Android package index. *developer.android.com/reference/packages*, Jul 2017.

[10] Android. Android platform frameworks base. *github.com/android/*, Aug 2017.

[11] G Antoniol, M. Di Penta, and E Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, pages 31–40, USA, sep 2004. IEEE.

[12] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, mar 2007.

[13] Ron Artstein and Massimo Poesio. Inter-coder agreement for computational linguistics. *Computational Linguistics*, 34(4):555–596, 2008.

[14] Muhammad Asaduzzaman, Chanchal K. Roy, Samiul Monir, and Kevin A. Schneider. Exploring API method parameter recommendations. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 271–280, USA, sep 2015. IEEE.

[15] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, ICSE '19, pages 748–758, Piscataway, NJ, USA, may 2019. IEEE.

[16] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. An exploratory study on faults in web API integration in a large-scale payment

company. In *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18*, pages 13–22, New York, New York, USA, 2018. ACM Press.

[17] Shams Azad, Peter C. Rigby, and Latifa Guerrouj. Generating API Call Rules from Version History and Stack Overflow Posts. *ACM Transactions on Software Engineering and Methodology*, 25(4):1–22, jan 2017.

[18] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE 10*, 2010.

[19] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. Analyzing a decade of Linux system calls. *Empirical Software Engineering*, 23(3):1519–1551, jun 2018.

[20] Mehdi Bahrami, Junhee Park, Lei Liu, and Wei-Peng Chen. API Learning. In *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*, pages 151–154, New York, New York, USA, 2018. ACM Press.

[21] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. *ACM SIGPLAN Notices*, 40(10):265, oct 2005.

[22] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs van der Storm. Study of an API Migration for Two XML APIs. In *Software Language Engineering. SLE 2009*, pages 42–61. Springer, Berlin, Heidelberg, 2010.

[23] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the apache community upgrades dependencies: An evolutionary study. *Empirical Softw. Engg.*, 20:1275–1317, October 2015.

[24] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, apr 2015.

[25] Wayne Beaton. Eclipse corner article abstract syntax tree, the eclipse foundation, 2019.

[26] Kent Beck. Junit. *JUnit.org*, Jul 2019.

[27] Boualem Benatallah and Fabio Casati. Panel on Cognitive Service Engineering. In *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*, pages 883–883, New York, New York, USA, 2018. ACM Press.

[28] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R Motahari Nezhad, and Farouk Toumani. Developing Adapters for Web Services Integration. In Oscar Pastor and João e Cunha, editors, *Advanced Information Systems Engineering*, pages 415–429, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[29] David Bermbach and Erik Wittern. Benchmarking Web API Quality. In *Web Engineering. ICWE 2016.*, pages 188–206. Springer, Berlin, Heidelberg, 2016.

[30] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 109–120, New York, NY, USA, 2016. ACM.

[31] Salah Bouktif, Houari Sahraoui, and Faheem Ahmed. Predicting Stability of Open-Source Software Systems Using Combination of Bayesian Classifiers. *ACM Transactions on Management Information Systems*, 5(1):1–26, apr 2014.

[32] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 2018-March, pages 507–511, USA, mar 2018. IEEE.

[33] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 2018-March, pages 255–265, USA, mar 2018. IEEE.

[34] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do Developers Deprecate APIs with Replacement Messages? A Large-Scale Analysis on Java Systems. In *2016 IEEE 23rd International Conference on Software*

Analysis, Evolution, and Reengineering (SANER), volume 1, pages 360–369, USA, mar 2016. IEEE.

[35] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software*, 137:306–321, mar 2018.

[36] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. SAR: learning cross-language API mappings with little knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, pages 796–806, New York, New York, USA, 2019. ACM Press.

[37] Raymond P. L. Buse and Westley Weimer. Synthesizing API usage examples. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 782–792, USA, jun 2012. IEEE.

[38] John Businge, Alexander Serebrenik, and M. van den Brand. Analyzing the Eclipse API Usage: Putting the Developer in the Loop. In *17th European Conference on Software Maintenance and Reengineering*, pages 37–46, USA, mar 2013. IEEE.

[39] Paolo Calciati, Konstantin Kuznetsov, Xue Bai, and Alessandra Gorla. What did really change with the new release of the app? In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, pages 142–152, New York, New York, USA, 2018. ACM Press.

[40] Joao Campinhos, Joao Costa Seco, and Jacome Cunha. Type-Safe Evolution of Web Services. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*, pages 20–26, USA, may 2017. IEEE.

[41] William B. Cavnar and John M. Trenkle. N-gram-based text categorization. In *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[42] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching connected api subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International*

Symposium on the Foundations of Software Engineering, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[43] Cong Chen and Kang Zhang. Who asked what: integrating crowdsourced FAQs into API documentation. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, ICSE Companion 2014, pages 456–459, New York, New York, USA, 2014. ACM Press.

[44] Yonghong Chen, Xiwei Xu, and Liming Zhu. Web Platform API Design Principles and Service Contract. In *2012 19th Asia-Pacific Software Engineering Conference*, volume 1, pages 877–886, USA, dec 2012. IEEE.

[45] Chow and Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of International Conference on Software Maintenance ICSM-96*, pages 359–368, USA, 1996. IEEE.

[46] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. *2013 IEEE International Conference on Software Maintenance*, 2013.

[47] Oracle Corporation. Java streams. *java.util.stream (Java Platform SE 8)*, Jul 2020.

[48] Bradley E Cossette and Robert J Walker. Seeking the ground truth. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, FSE '12, page 1, USA, 2012. ACM Press.

[49] Ira W. Cotton and Frank S. Greatorex, Jr. Data structures and techniques for remote computer graphics. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 533–544, New York, NY, USA, 1968. ACM.

[50] Barthélémy Dagenais and Harold Ossher. Automatically locating framework extension examples. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, SIGSOFT '08/FSE-16, page 203, New York, New York, USA, 2008. ACM Press.

[51] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 20(4):481, 2008.

[52] Barthelemy Dagenais and Martin P. Robillard. SemDiff: Analysis and recommendation support for API evolution. In *2009 IEEE 31st International Conference on Software Engineering*, pages 599–602, USA, 2009. IEEE.

[53] Barthélémy Dagenais and Martin P Robillard. Creating and evolving developer documentation. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*, FSE '10, page 127, New York, New York, USA, 2010. ACM Press.

[54] Barthelemy Dagenais and Martin P. Robillard. Recovering traceability links between an api and its learning resources. *2012 34th International Conference on Software Engineering (ICSE)*, 2012.

[55] Marco D'Ambros, Michele Lanza, Mircea Lungu, and Romain Robbes. Promises and perils of porting software visualization tools to the web. In *2009 11th IEEE International Symposium on Web Systems Evolution*, pages 109–118, USA, sep 2009. IEEE.

[56] Fernando López de la Mora and Sarah Nadi. Which library should i use? a metric-based comparison of software libraries. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, page 37–40, New York, NY, USA, 2018. Association for Computing Machinery.

[57] Coen De Roover, Ralf Lammel, and Ekaterina Pek. Multi-dimensional exploration of API usage. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 152–161, USA, may 2013. IEEE.

[58] Uri Dekel and James D. Herbsleb. Improving api documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, page 320–330, USA, 2009. IEEE Computer Society.

[59] Uri Dekel and James D. Herbsleb. Reading the documentation of invoked API functions in program comprehension. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 168–177, USA, may 2009. IEEE.

[60] Java design patterns.com. Java pipeline pattern. *java-design-patterns.com*, 2019.

[61] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, volume 1, pages 64–73, USA, feb 2014. IEEE.

[62] Danny Dig. Using refactorings to automatically update component-based applications. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '05*, page 234, New York, New York, USA, 2005. ACM Press.

[63] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 404–428. Springer-Verlag, Berlin, Heidelberg, 2006.

[64] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005:389–398, 2005.

[65] Danny Dig and Ralph Johnson. Automated upgrading of component-based applications. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, volume 2006, page 675, New York, New York, USA, 2006. ACM Press.

[66] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, mar 2006.

[67] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proceedings*

of the 29th International Conference on Software Engineering, ICSE '07, page 427–436, USA, 2007. IEEE Computer Society.

[68] Ekwa Duala-Ekoko and Martin P. Robillard. Using Structure-Based Recommendations to Facilitate Discoverability in APIs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6813 LNCS, pages 79–104, USA, 2011. Springer.

[69] Ekwa Duala-Ekoko and Martin P Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 266–276, USA, jun 2012. IEEE.

[70] Anna Maria Eilertsen and Anya Helene Bagge. Exploring API. In *Proceedings of the 2nd International Workshop on API Usage and Evolution - WAPI '18*, pages 10–13, New York, New York, USA, 2018. ACM Press.

[71] Daniel S. Eisenberg, Jeffrey Stylos, and Brad A. Myers. Apatite. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, page 1331, New York, New York, USA, 2010. ACM Press.

[72] Renée Elio, Eleni Stroulia, and Warren Blanchet. Using Interaction Models to Detect and Resolve Inconsistencies in Evolving Service Compositions. *Web Intelli. and Agent Sys.*, 7(2):139–160, apr 2009.

[73] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th International Conference on Software Engineering*, page 302–312, USA, 2007. IEEE Computer Society.

[74] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do API documentation and static typing affect API usability? In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 632–642, New York, New York, USA, 2014. ACM Press.

[75] M.D. Ernst, Jake Cockrell, W.G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[76] Tiago Espinha, Andy Zaidman, and Hans-Gerhard 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, USA, feb 2014. IEEE.

[77] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web api fragility: How robust is your mobile application? In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '15, page 12–21, USA, 2015. IEEE Press.

[78] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*, 100:27–43, feb 2015.

[79] F-Droid. F-droid, a free and open source android app repository. *F-Droid - Free and Open Source Android App Repository*, 2017.

[80] FasterXML. Fasterxml/jackson. *Jackson*, Jul 2019.

[81] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 791–796, New York, New York, USA, 2018. ACM Press.

[82] The Apache Software Foundation. Apache maven, Jul 2019.

[83] The Apache Software Foundation. Log4j. *The Apache Software Foundation*, Jul 2019.

[84] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 254–265, New York, NY, USA, 2016. Association for Computing Machinery.

[85] Tammo Freese. Inline Method Considered Helpful: An Approach to Interface Evolution. In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering*, XP'03, pages 271–278. Springer-Verlag, Berlin, Heidelberg, 2003.

[86] Tammo Freese. Refactoring-aware version control. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, volume 2006, page 953, New York, New York, USA, 2006. ACM Press.

[87] Davide Fucci, Alireza Mollaalizadehbahnemiri, and Walid Maalej. On using machine learning to identify knowledge in API reference documentation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, pages 109–119, New York, New York, USA, 2019. ACM Press.

[88] Jun Gao, Pingfan Kong, Li Li, Tegawende F. Bissyande, and Jacques Klein. Negative Results on Mining Crypto-API Usage Rules in Android Apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 388–398, USA, may 2019. IEEE.

[89] Simos Gerasimou, Maria Kechagia, Dimitris Kolovos, Richard Paige, and Georgios Gousios. On software modernisation due to library obsolescence. In *Proceedings of the 2nd International Workshop on API Usage and Evolution - WAPI '18*, pages 6–9, New York, New York, USA, 2018. ACM Press.

[90] Bashar Gharaibeh, Tien N. Nguyen, and J. Morris Chang. Coping with API Evolution for Running, Mission-Critical Applications Using Virtual Execution Environment. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 171–180, USA, 2007. IEEE.

[91] Elena L Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. Visualizing API Usage Examples at Scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, pages 1–12, USA, 2018. ACM Press.

[92] M.W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, feb 2005.

[93] Google. Google samples, android developers. *Android Developers*, 2019.

[94] Google. Guava, google core libraries for java. *GitHub.com/Google/Guava*, Jul 2019.

[95] William Granli, John Burchell, Imed Hammouda, and Eric Knauss. The driving forces of API evolution. In *Proceedings of the 14th International Workshop on Principles of Software Evolution - IWPSE 2015*, volume 30-Aug-201, pages 28–37, New York, New York, USA, 2015. ACM Press.

[96] Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado A. Visaggio, Gerardo Canfora, and Sebastiano Panichella. Android apps and user feedback: a dataset for software evolution and quality improvement. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics - WAMA 2017*, pages 8–11, New York, New York, USA, 2017. ACM Press.

[97] Andrew F. Hayes and Klaus Krippendorff. Answering the call for a standard reliability measure for coding data. *Communication Methods and Measures*, 1:77–89, 2007.

[98] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 167–177, New York, New York, USA, 2018. ACM Press.

[99] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 1663–1671, New York, New York, USA, 2014. ACM Press.

[100] J. Henkel and A. Diwan. Catchup! capturing and replaying refactorings to support API evolution. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 274–283, USA, 2005. IEEE.

[101] Robert Heumüller, Sebastian Nielebock, and Frank Ortmeier. Who plays with whom? ... and how? mining API interaction patterns from source code. In *Proceedings of the 7th International Workshop on Software Mining - SoftwareMining 2018*, pages 8–11, New York, New York, USA, 2018. ACM Press.

[102] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 117–125, 2005.

[103] Reid Holmes and Robert J. Walker. A newbie's guide to eclipse APIs. In *Proceedings of the 2008 international workshop on Mining software repositories - MSR '08*, page 149, New York, New York, USA, 2008. ACM Press.

[104] André Hora, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Túlio Valente. Automatic detection of system-specific conventions unknown to developers. *Journal of Systems and Software*, 109:192–204, nov 2015.

[105] Andre Hora, Anne Etien, Nicolas Anquetil, Stephane Ducasse, and Marco Tulio Valente. APIEvolutionMiner: Keeping API evolution under control. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 420–424, USA, feb 2014. IEEE.

[106] Andre Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Tulio Valente. How do developers react to API evolution? The Pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, volume 3, pages 251–260, USA, sep 2015. IEEE.

[107] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal*, 26(1):161–191, mar 2018.

[108] Andre Hora and Marco Tulio Valente. Apiwave: Keeping track of API popularity and migration. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 321–323, USA, sep 2015. IEEE.

[109] André Hora, Marco Tulio Valente, Romain Robbes, and Nicolas Anquetil. When should internal interfaces be promoted to public? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, volume 13-18-Nove, pages 278–289, New York, New York, USA, 2016. ACM Press.

[110] Daqing Hou and Lin Li. Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 91–100, USA, jun 2011. IEEE.

[111] Daqing Hou and Xiaojia Yao. Exploring the Intent behind API Evolution: A Case Study. In *2011 18th Working Conference on Reverse Engineering*, pages 131–140, USA, oct 2011. IEEE.

[112] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 293–304, New York, New York, USA, 2018. ACM Press.

[113] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia. A survey on adaptive random testing. *IEEE Transactions on Software Engineering*, 1(1):1–1, 2019.

[114] Shiyou Huang, Jianmei Guo, Sanhong Li, Xiang Li, Yumin Qi, Kingsum Chow, and Jeff Huang. SafeCheck: Safety Enhancement of Java Unsafe API. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, volume 2019-May, pages 889–899, USA, may 2019. IEEE.

[115] Jan Hýbl and Zdeněk Troníček. On testing the source compatibility in Java. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity - SPLASH '13*, pages 87–88, New York, New York, USA, 2013. ACM Press.

[116] Andrew J. Ko, B.A. Myers, and H.H. Aung. Six Learning Barriers in End-User Programming Systems. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 199–206, USA, 2004. IEEE.

[117] JavaParser. Javaparser. *JavaParser.org*, Jul 2019.

[118] J. Jiang, Johannes Koskinen, Anna Ruokonen, and T. Systa. Constructing Usage Scenarios for API Redocumentation. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 259–264, USA, jun 2007. IEEE.

[119] Z. M. Jiang and A. E. Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

[120] Johnson. Substring matching for clone detection and change tracking. In *Proceedings International Conference on Software Maintenance ICSM-94*, pages 120–126, USA, 1994. IEEE Comput. Soc. Press.

[121] Sukrit Kalra, Ayush Goel, Dhriti Khanna, Mohan Dhawan, Subodh Sharma, and Rahul Purandare. POLLUX: safely upgrading dependent application libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, volume 13-18-Nove, pages 290–300, New York, New York, USA, 2016. ACM Press.

[122] Puneet Kapur, Brad Cossette, and Robert J Walker. Refactoring references for library migration. *ACM SIGPLAN Notices*, 45(10):726, oct 2010.

[123] R H Katz. The post-PC era. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications - DIALM '00*, page 64, New York, New York, USA, 2000. ACM Press.

[124] David Kawrykow and Martin P. Robillard. Improving API Usage through Automatic Detection of Redundant Code. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 111–122, USA, nov 2009. IEEE.

[125] Jungil Kim and Eunjoo Lee. The effect of IMPORT change in software change history. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 1753–1754, New York, New York, USA, 2014. ACM Press.

[126] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 151–160, New York, NY, USA, 2011. Association for Computing Machinery.

[127] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 309–319, USA, 2009. IEEE.

[128] Miryung Kim, David Notkin, and Dan Grossman. Automatic Inference of Structural Changes for Matching across Program Versions. In *29th International Conference on Software Engineering (ICSE'07)*, ICSE '07, pages 333–343, Washington, DC, USA, may 2007. IEEE.

[129] B. Kitchenham and S Charters. Guidelines for performing systematic literature reviews in software engineering, Keele University, Keele, U.K., Tech. Rep. EBSE-2007-01, 2007.

[130] Deokyoon Ko, Kyeongwook Ma, Sooyong Park, Suntae Kim, Dongsun Kim, and Yves Le Traon. API Document Quality for Resolving Deprecated APIs. In *21st Asia-Pacific Software Engineering Conference*, volume 2, pages 27–30, USA, dec 2014. IEEE.

[131] Klaus Krippendorff. Computing krippendorffs alpha reliability. *University of Pennsylvania Scholarly Commons*, Jan 2011.

[132] Klaus H. Krippendorff. *Content Analysis - 3rd Edition: an Introduction to Its Methodology*. SAGE Publications, Inc, 2013.

[133] Raula Gaikovina Kula, Ali Ouni, Daniel M. German, and Katsuro Inoue. An empirical study on the impact of refactoring activities on evolving client-used APIs. *Information and Software Technology*, 93(July 2016):186–199, jan 2018.

[134] Hobum Kwon, Juwon Ahn, Sunggyu Choi, Jakub Siewierski, Piotr Kosko, and Piotr Szydelko. An Experience Report of the API Evolution and Maintenance for Software Platforms. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 587–590, USA, sep 2018. IEEE.

[135] Maxime Lamothe and Weiyi Shang. Exploring the use of automated API migrating techniques in practice. *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, 1:503–514, 2018.

[136] Maxime Lamothe and Weiyi Shang. When apis are intentionally bypassed: An exploratory study of api workarounds. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 912–924, New York, NY, USA, 2020. Association for Computing Machinery.

[137] Craig Larman. Protected variation: the importance of being closed. *IEEE Software*, 18(3):89–91, 2001.

[138] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 602–613, New York, NY, USA, 2016. Association for Computing Machinery.

[139] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[140] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, page 108–124, Berlin, Heidelberg, 1996. Springer-Verlag.

[141] Grace A. Lewis and Dennis B. Smith. Service-Oriented Architecture and its implications for software maintenance and evolution. In *2008 Frontiers of Software Maintenance*, pages 1–10, USA, sep 2008. IEEE.

[142] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193, 2018.

[143] Jing Li, Aixin Sun, Zhenchang Xing, and Lei Han. API Caveat Explorer – Surfacing Negative Usages from Practice. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval - SIGIR '18*, pages 1293–1296, New York, New York, USA, 2018. ACM Press.

[144] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How Does Web Service API Evolution Affect Clients? *2013 IEEE 20th International Conference on Web Services*, 1:300–307, jun 2013.

[145] Li Li and Wu Chou. Designing Large Scale REST APIs Based on REST Chart. In *2015 IEEE International Conference on Web Services*, ICWS '15, pages 631–638, Washington, DC, USA, jun 2015. IEEE.

[146] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, pages 254–264, New York, New York, USA, 2018. ACM Press.

[147] Mario Linares-Vásquez. Supporting evolution and maintenance of Android apps. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 714–717, New York, New York, USA, 2014. ACM Press.

[148] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API change and fault proneness: a threat to the success of Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 477, New York, New York, USA, 2013. ACM Press.

[149] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: an empirical study. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, 1:2–11, 2014.

[150] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do API changes trigger stack overflow discussions? a study on the Android SDK. In *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*, pages 83–94, New York, New York, USA, 2014. ACM Press.

[151] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. Generating query-specific class API summaries. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, ESEC/FSE 2019, pages 120–130, New York, New York, USA, 2019. ACM Press.

[152] Xiaoyu Liu, LiGuo Huang, and Vincent Ng. Effective API recommendation without historical software repositories. In *Proceedings of the 33rd ACM/IEEE*

International Conference on Automated Software Engineering - ASE 2018, pages 282–292, New York, New York, USA, 2018. ACM Press.

[153] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 415–425, New York, NY, USA, 2015. ACM.

[154] Homan Ma, Robert Amor, and Ewan Tempero. Indexing the Java API Using Source Code. In *19th Australian Conference on Software Engineering (aswec 2008)*, pages 451–460, USA, mar 2008. IEEE.

[155] Walid Maalej and Martin P. Robillard. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, sep 2013.

[156] Andrew Macvean, Martin Maly, and John Daughtry. Api design reviews at scale. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '16, page 849–858, New York, NY, USA, 2016. Association for Computing Machinery.

[157] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. In *2010 Eighth IEEE European Conference on Web Services*, pages 107–114, USA, dec 2010. IEEE.

[158] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 48–61, New York, NY, USA, 2005. Association for Computing Machinery.

[159] Konstantinos Manikas. Revisiting software ecosystems Research: A longitudinal literature study. *Journal of Systems and Software*, 117:84–103, jul 2016.

[160] Robert Cecil. Martin. *Clean code*. Apogeo, 2018.

[161] Anderson S. Matos, Joao B. Ferreira Filho, and Lincoln S. Rocha. Splitting APIs: An Exploratory Study of Software Unbundling. In *IEEE/ACM 16th International Conference on Mining Software Repositories*, pages 360–370, USA, may 2019. IEEE.

[162] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79, USA, sep 2013. IEEE.

[163] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. Recommending source code examples via api call usages and documentation. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, page 21–25, New York, NY, USA, 2010. Association for Computing Machinery.

[164] Na Meng, Miryung Kim, and Kathryn S. McKinley. Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 440–443, New York, NY, USA, 2011. ACM.

[165] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, page 502–511, USA, 2013. IEEE Press.

[166] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 353–363, USA, jun 2012. IEEE.

[167] Kim Mens and Angela Lozano. Source Code-Based Recommendation Systems. In *Recommendation Systems in Software Engineering*, pages 93–130. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[168] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, may 2002.

[169] Gebremariam Mesfin, Tor-Morten Grønli, Dida Midekso, and Gheorghita Ghinca. Towards end-user development of REST client applications on smartphones. *Computer Standards & Interfaces*, 44:205–219, feb 2016.

[170] A. Michail. Data mining library reuse patterns in user-selected applications. In *14th IEEE International Conference on Automated Software Engineering*, pages 24–33, USA, 2003. IEEE Comput. Soc.

[171] Microsoft. interface c# reference 2019. *Docs.microsoft.com*, 2019.

[172] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining API Popularity. In *Testing – Practice and Research Techniques. TAIC PART 2010*, pages 173–180. Springer, Berlin, Heidelberg, 2010.

[173] Yana Momchilova Mileva, Andrzej Wasylkowski, and Andreas Zeller. Mining Evolution of Object Usage. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, pages 105–129. Springer-Verlag, Berlin, Heidelberg, 2011.

[174] Joao Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *20th Working Conference on Reverse Engineering*, pages 401–408, USA, oct 2013. IEEE.

[175] Brandon Morel and Perry Alexander. SPARTACAS: automating component reuse and adaptation. *IEEE Transactions on Software Engineering*, 30(9):587–600, sep 2004.

[176] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How Can I Use This Method? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 880–890, USA, may 2015. IEEE.

[177] Laura Moreno and Andrian Marcus. Automatic software summarization: the state of the art. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[178] Dan Morrill. Announcing the android 1.0 sdk, release 1, Jan 1970.

[179] Eduardo Mosqueira-Rey, David Alonso-Ríos, Vicente Moret-Bonillo, Isaac Fernández-Varela, and Diego Álvarez-Estévez. A systematic approach to API usability: Taxonomy-derived criteria and a case study. *Information and Software Technology*, 97(December 2017):46–63, may 2018.

[180] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017*, pages 215–225, New York, New York, USA, 2017. ACM Press.

[181] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers. Api designers in the field: Design practices and challenges for creating usable apis. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 249–258, 2018.

[182] Emerson Murphy-Hill, Caitlin Sadowski, Andrew Head, John Daughtry, Andrew Macvean, Ciera Jaspan, and Collin Winter. Discovering API usability problems at scale. In *Proceedings of the 2nd International Workshop on API Usage and Evolution - WAPI '18*, pages 14–17, New York, New York, USA, 2018. ACM Press.

[183] Brad A. Myers. Human-Centered Methods for Improving API Usability. In *2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*, volume 49, pages 2–2, USA, may 2017. IEEE.

[184] Varvana Myllärniemi, Sari Kujala, Mikko Raatikainen, and Piia Sevonn. Development as a journey: factors supporting the adoption and use of software frameworks. *Journal of Software Engineering Research and Development*, 6(1):6, dec 2018.

[185] D. Nam, A. Horvath, A. Macvean, B. Myers, and B. Vasilescu. Marble: Mining for boilerplate code to identify api usability problems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 615–627, 2019.

[186] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522, USA, 2016. ACM Press.

[187] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 457–468, New York, New York, USA, 2014. ACM Press.

[188] Anh Tuan Nguyen, Peter C. Rigby, Thanh Van Nguyen, Mark Karanfil, and Tien N. Nguyen. Statistical Translation of English Texts to API Code Templates. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 331–333, USA, may 2017. IEEE.

[189] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 166–177, New York, New York, USA, 2014. ACM Press.

[190] Hoan Anh Nguyen, Tien N. Nguyen, Hridesh Rajan, and Robert Dyer. Towards combining usage mining and implementation analysis to infer API preconditions. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Automated Specification Inference - WASPI 2018*, pages 15–16, New York, New York, USA, 2018. ACM Press.

[191] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. *ACM SIGPLAN Notices*, 45(10):302, oct 2010.

[192] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In *2019 IEEE/ACM 41st*

International Conference on Software Engineering (ICSE), pages 1050–1060, USA, may 2019. IEEE.

[193] Thanh Nguyen, Peter C. Rigby, Anh Tuan Nguyen, Mark Karanfil, and Tien N. Nguyen. T2API: synthesizing API code usage templates from English texts with statistical translation. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1013–1017, USA, 2016. ACM Press.

[194] Thanh Nguyen, Ngoc Tran, Hung Phan, Trong Nguyen, Linh Truong, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Complementing global and local contexts in representing API descriptions to improve API retrieval tasks. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE*, pages 551–562, USA, 2018. ACM Press.

[195] Thanh V. Nguyen and Tien N. Nguyen. Inferring API elements relevant to an english query. In *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE '18*, volume Part F1373, pages 167–168, New York, New York, USA, 2018. ACM Press.

[196] Tien N. Nguyen. Code migration with statistical machine translation. In *Proceedings of the 5th International Workshop on Software Mining - SoftwareMining 2016*, pages 2–2, New York, New York, USA, 2016. ACM Press.

[197] T.N. Nguyen, E.V. Munson, and C. Thao. Managing the evolution of Web-based applications with WebSCM. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 577–586, USA, 2005. IEEE.

[198] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Mapping API elements for code migration with vector representations. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, ICSE '16, pages 756–758, New York, New York, USA, 2016. ACM Press.

[199] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. Exploring API Embedding for API Usages and Applications. In *2017*

IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 438–449, USA, may 2017. IEEE.

[200] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 383–392, New York, NY, USA, 2009. Association for Computing Machinery.

[201] Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 1, page 205, New York, New York, USA, 2010. ACM Press.

[202] Team NLTK. Natural language toolkit. *NLTK.org*, 2019.

[203] Oracle. what is an interface? (java™ learning the java language object-oriented programming concepts, 2019.

[204] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[205] Chris Parnin and Christoph Treude. Measuring API documentation on the web. In *Proceeding of the 2nd international workshop on Web 2.0 for software engineering - Web2SE '11*, pages 25–30, New York, New York, USA, 2011. ACM Press.

[206] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. *Georgia Tech Technical Report*, 1:1–11, 2012.

[207] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Inf. Softw. Technol.*, 64(C):1–18, August 2015.

[208] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE:

Java vs. groovy with eclipse. In *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*, pages 212–222, New York, New York, USA, 2014. ACM Press.

[209] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen. Statistical migration of api usages. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 47–50, May 2017.

[210] Hung Phan, Hoan Anh Nguyen, Ngoc M Tran, Linh H Truong, Anh Tuan Nguyen, and Tien N. Nguyen. Statistical learning of API fully qualified names in code snippets of online forums. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, volume 11, pages 632–642, New York, New York, USA, 2018. ACM Press.

[211] Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. An Empirical Study of API Usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 5–14, USA, oct 2013. IEEE.

[212] David M Pletcher and Daqing Hou. BCC: Enhancing code completion for better API usability. *2009 IEEE International Conference on Software Maintenance*, 1:393–394, sep 2009.

[213] Shankar R Ponnekanti and Armando Fox. Interoperability Among Independently Evolving Web Services. In *5th ACM/IFIP/USENIX International Conference on Middleware, MIDDLEWARE 2004*, volume LNCS 3231, pages 331–351. Springer-Verlag, Berlin, Heidelberg, 2004.

[214] Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. Java reflection API: revealing the dark side of the mirror. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, pages 636–646, New York, New York, USA, 2019. ACM Press.

[215] Ivan Porres and Irum Rauf. Modeling behavioral RESTful web service interfaces in UML. In *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*, SAC '11, page 1598, New York, New York, USA, 2011. ACM Press.

[216] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 91–100, 2014.

[217] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 925–935, USA, jun 2012. IEEE.

[218] Lutz Prechelt and D.J. Hutzel. The co-evolution of a hype and a software architecture: experience of component-producing large-scale EJB early adopters. In *25th International Conference on Software Engineering, 2003. Proceedings.*, volume 0, pages 553–556, USA, 2003. IEEE.

[219] Qi Xi, Tianyang Zhou, Qingxian Wang, and Yongjun Zeng. An API deobfuscation method combining dynamic and static techniques. *Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC)*, 1:2133–2138, dec 2013.

[220] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 378–387, USA, sep 2012. IEEE.

[221] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. RACK: Automatic API Recommendation Using Crowdsourced Knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 349–359, USA, mar 2016. IEEE.

[222] RedHat. Hibernate ogm. *Hibernate.org*, Jul 2019.

[223] Anastasia Reinhardt, Tianyi Zhang, Mihir Mathur, and Miryung Kim. Augmenting stack overflow with API usage patterns mined from GitHub. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 880–883, USA, 2018. ACM Press.

[224] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. Api-misuse detection driven by fine-grained api-constraint knowledge graph. *35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[225] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, Dec 2010.

[226] Romain Robbes and Mircea Lungu. A study of ripple effects in software ecosystems. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, ICSE '11, page 904, New York, New York, USA, 2011. ACM Press.

[227] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 1, USA, 2012. ACM.

[228] Martin P Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26:27–34, 2009.

[229] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, may 2013.

[230] Martin P Robillard and Yam B Chhetri. Recommending reference API documentation. *Empirical Software Engineering*, 20(6):1558–1586, dec 2015.

[231] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, dec 2011.

[232] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[233] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurelio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vasquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. On-demand Developer Documentation. *2017 IEEE*

International Conference on Software Maintenance and Evolution (ICSME), 1:479–483, sep 2017.

[234] D Rose, S Stegmaier, G Reina, D Weiskopf, and T Ertl. Non-invasive Adaptation of Black-box User Interfaces. *Reproduction*, 1:1–1, 2002.

[235] Gordon Rugg and Peter Mcgeorge. The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts. *Expert Systems*, 14(2):80–93, 1997.

[236] Thomas Ruhroth and Heike Wehrheim. Refinement-preserving co-evolution. In *Formal Methods and Software Engineering. ICFEM 2009.*, pages 620–638. Springer, Berlin, Heidelberg, 2009.

[237] Chandan R Rupakheti and Daqing Hou. Evaluating forum discussions to inform the design of an API critic. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 53–62, USA, jun 2012. IEEE.

[238] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. An observational study on API usage constraints and their documentation. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 1:33–42, mar 2015.

[239] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, and Filomena Ferrucci. Do developers update third-party libraries in mobile apps? In *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*, pages 255–265, New York, New York, USA, 2018. ACM Press.

[240] Satista. App stores: number of apps in leading app stores 2016, 2017.

[241] Anand Ashok Sawant, Maurício Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, volume 11, pages 561–571, New York, New York, USA, 2018. ACM Press.

[242] Anand Ashok Sawant and Alberto Bacchelli. A Dataset for API Usage. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, volume 2015-Augus, pages 506–509, USA, may 2015. IEEE.

[243] Anand Ashok Sawant and Alberto Bacchelli. fine-GRAPE: fine-grained APi usage extractor – an approach and dataset to investigate API usage. *Empirical Software Engineering*, 22(3):1348–1371, jun 2017.

[244] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK. *Empirical Software Engineering*, 23(4):2158–2197, aug 2018.

[245] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vasquez, Michele Lanza, and Rocco Oliveto. Data-Driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, volume 2019-May, pages 288–298, USA, may 2019. IEEE.

[246] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 13th international conference on Software engineering*, ICSE, page 471, USA, 2008. ACM Press.

[247] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. An Empirical Study on Evolution of API Documentation. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, pages 416–431. Springer-Verlag, Berlin, Heidelberg, 2011.

[248] S M Sohan, Craig Anslow, and Frank Maurer. SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server. In *30th IEEE/ACM International Conference on Automated Software Engineering*, pages 271–276, USA, nov 2015. IEEE.

[249] S.M. Sohan, Craig Anslow, and Frank Maurer. A Case Study of Web API Evolution. In *2015 IEEE World Congress on Services*, pages 245–252, USA, jun 2015. IEEE.

[250] Brett Spell. *Pro Java 8 programming*. Apress, USA, 2015.

[251] Stack exchange data dump : Stack exchange, inc. : Free download, borrow, and streaming.

[252] Statista. Android statistics and facts. *Statista.com*, 2017.

[253] Roman Štrobl and Zdeněk Troníček. Migration from deprecated API in Java. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity - SPLASH '13*, SPLASH '13, pages 85–86, New York, New York, USA, 2013. ACM Press.

[254] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving API documentation using API usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126, USA, sep 2009. IEEE.

[255] Jeffrey Stylos, Benjamin Graf, Daniela K. Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. A case study of API redesign for improved usability. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 189–192, USA, sep 2008. IEEE.

[256] Jeffrey Stylos and B.A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 195–202, USA, 2006. IEEE.

[257] Jeffrey Stylos and Brad A. Myers. The implications of method placement on api learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 105–112, New York, NY, USA, 2008. Association for Computing Machinery.

[258] Jingyi Su, Mohd Arafat, and Robert Dyer. Using consensus to automatically infer post-conditions. In *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings*, pages 202–203, USA, 2018. ACM Press.

[259] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 643–652, New York, New York, USA, 2014. ACM Press.

[260] Sunghun Kim, Kai Pan, and E.J. Whitehead. When functions change their names: automatic detection of origin relationships. In *12th Working Conference*

on Reverse Engineering (WCRE'05), volume 2005, pages 10 pp.–152, USA, 2005. IEEE.

[261] Philippe Suter and Erik 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, USA, nov 2015. IEEE.

[262] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of api refactorings in libraries. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, page 377, New York, New York, USA, 2007. ACM Press.

[263] the Apache Software Foundation. Apache lucene. *lucene.apache.org*, 2019.

[264] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336, USA, sep 2008. IEEE.

[265] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. Automatic recommendation of API methods from feature requests. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1:290–300, nov 2013.

[266] T. Tourwe and Tom Mens. Automated support for framework-based software. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 148–157, USA, 2003. IEEE Comput. Soc.

[267] Christoph Treude and Maurício Aniche. Where does Google find API documentation? *Proceedings of the 2nd International Workshop on API Usage and Evolution - WAPI '18*, 2:19–22, 2018.

[268] Christoph Treude and Martin P. Robillard. Augmenting API documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, volume 14-22-May-, pages 392–403, New York, New York, USA, 2016. ACM Press.

[269] Zdene Troni. API Evolution with RefactoringNG. In *2010 Second World Congress on Software Engineering*, volume 2, pages 293–297, USA, dec 2010. IEEE.

[270] Zdeněk Troníček. RefactoringNG. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, page 1165, New York, New York, USA, 2012. ACM Press.

[271] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. Softw.*, 84(10):1757–1782, October 2011.

[272] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, pages 483–494, New York, New York, USA, 2018. ACM Press.

[273] Romanos Tsouroplis, Michael Petychakis, Iosif Alvertis, Evmorfia Biliri, Fenareti Lampathaki, and Dimitris Askounis. Internet-Based Enterprise Innovation Through a Community-Based API Builder to Manage APIs. In *Current Trends in Web Engineering. ICWE 2015.*, pages 65–76. Springer, Berlin, Heidelberg, 2015.

[274] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, 2016.

[275] Gias Uddin, Barthelemy Dagenais, and Martin P. Robillard. Analyzing temporal API usage patterns. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 456–459, USA, nov 2011. IEEE.

[276] Gias Uddin and Martin P. Robillard. How API Documentation Fails. *IEEE Software*, 32(4):68–75, jul 2015.

[277] SungyYong Um. The evolution of a digital ecosystem. In *Companion to the Proceedings of the 11th International Symposium on Open Collaboration - OpenSym '15*, pages 1–1, New York, New York, USA, 2015. ACM Press.

[278] Thanh Van Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Characterizing API elements in software documentation with vector representation. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pages 749–751, New York, New York, USA, 2016. ACM Press.

[279] Pradeep K. Venkatesh, Shaohua Wang, Feng Zhang, Ying Zou, and Ahmed E. Hassan. What Do Client Developers Concern When Using Web APIs? An Empirical Study on Developer Forums and Stack Overflow. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 131–138, USA, jun 2016. IEEE.

[280] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei 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, USA, may 2013. IEEE.

[281] Shaohua Wang, Iman Keivanloo, and Ying Zou. How Do Developers React to RESTful API Evolution? *12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings*, 8831:245–259, 2014.

[282] Shaohua Wang, Nhathai Phan, Yan Wang, and Yong Zhao. Extracting API Tips from Developer Question and Answer Websites. In *IEEE/ACM 16th International Conference on Mining Software Repositories*, volume 1, pages 321–332, USA, may 2019. IEEE.

[283] Wei Wang and Michael W Godfrey. Detecting API usage obstacles: A study of iOS and Android developer questions. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 61–64, USA, may 2013. IEEE.

[284] Lili Wei, Yepang Liu, and Shing-Chi Cheung. PIVOT: Learning API-Device Correlations to Facilitate Android Compatibility Issue Detection. In *IEEE/ACM 41st International Conference on Software Engineering*, pages 878–888, USA, may 2019. IEEE.

[285] Tal Weiss. We analyzed 30,000 github projects - here are the top 100 libraries in java, js and ruby, Jan 2019.

[286] Peter Weissgerber and Stephan Diehl. Identifying Refactorings from Source-Code Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, USA, sep 2006. IEEE.

[287] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exposing Library API Misuses Via Mutation Analysis. In *IEEE/ACM 41st International Conference on Software Engineering*, volume 1, pages 866–877, USA, may 2019. IEEE.

[288] Erik Wilde. Surfing the API Web. In *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*, pages 797–803, New York, New York, USA, 2018. ACM Press.

[289] Titus Winters. Non-atomic refactoring and software sustainability. In *Proceedings of the 2nd International Workshop on API Usage and Evolution - WAPI '18*, pages 2–5, New York, New York, USA, 2018. ACM Press.

[290] Erik Wittern. Web APIs - challenges, design points, and research opportunities. In *Proceedings of the 2nd International Workshop on API Usage and Evolution - WAPI '18*, pages 18–18, New York, New York, USA, 2018. ACM Press.

[291] Erik Wittern, Annie T.T. Ying, Yunhui Zheng, Julian Dolby, and Jim A. Laredo. Statically Checking Web API Requests in JavaScript. In *IEEE/ACM 39th International Conference on Software Engineering*, pages 244–254, USA, may 2017. IEEE.

[292] Erik Wittern, Annie T.T. Ying, Yunhui Zheng, Jim A Laredo, Julian Dolby, Christopher C Young, and Aleksander A Slominski. Opportunities in Software Engineering Research for Web API Consumption. *2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*, 1:7–10, may 2017.

[293] Wei Wu. Modeling Framework API Evolution as a Multi-objective Optimization Problem. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 262–265, USA, jun 2011. IEEE.

[294] Wei Wu, Bram Adams, Yann-Gael Gueheneuc, and Giuliano Antoniol. ACUA: API Change and Usage Auditor. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 89–94, USA, sep 2014. IEEE.

[295] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, page 325, New York, New York, USA, 2010. ACM Press.

[296] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: A hybrid approach to identify framework evolution. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010.

[297] Wei Wu, Adrien Serveaux, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. The impact of imperfect change rules on framework API evolution identification: an empirical study. *Empirical Software Engineering*, 20:1126–1158, 2015.

[298] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147, USA, feb 2017. IEEE.

[299] Tao Xie and Jian Pei. MAPO. In *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, page 54, New York, New York, USA, 2006. ACM Press.

[300] Guowei Yang, Jeffrey Jones, Austin Moninger, and Meiru Che. How do Android operating system updates impact apps? In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems - MOBILESoft '18*, pages 156–160, New York, New York, USA, 2018. ACM Press.

[301] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. Learning to Extract API Mentions from Informal Natural Language Discussions. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 1:389–399, oct 2016.

[302] Reishi Yokomori, Harvey Siy, Masami Noro, and Katsuro Inoue. Assessing the impact of framework changes using component ranking. In *2009 IEEE*

International Conference on Software Maintenance, pages 189–198, USA, sep 2009. IEEE.

[303] Ping Yu, Fei Yang, Chun Cao, Hao Hu, and Xiaoxing Ma. API Usage Change Rules Mining based on Fine-grained Call Dependency Analysis. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware - Internetware'17*, volume Part F1309, pages 1–9, New York, New York, USA, 2017. ACM Press.

[304] Weizhao Yuan, Hoang H. Nguyen, Lingxiao Jiang, and Yuting Chen. LibraryGuru. In *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings*, pages 364–365, USA, 2018. ACM Press.

[305] Apostolos V. Zarras, Panos Vassiliadis, and Ioannis Dinos. Keep calm and wait for the spike! insights on the evolution of amazon services. In Selmin Nurcan, Pnina Soffer, Marko Bajec, and Johann Eder, editors, *CAiSE*, volume 9694 of *Lecture Notes in Computer Science*, pages 444–458. Springer International Publishing, Cham, 2016.

[306] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 17–23, New York, NY, USA, 2011. ACM.

[307] Amir Zghidi, Imed Hammouda, Brahim Hnich, and Eric Knauss. On the Role of Fitness Dimensions in API Design Assessment - An Empirical Investigation. In *2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*, pages 19–22, USA, may 2017. IEEE.

[308] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical API usage. In *34th International Conference on Software Engineering*, pages 826–836, USA, jun 2012. IEEE.

[309] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. Are code examples on an online Q&A forum reliable? In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, pages 886–896, New York, New York, USA, 2018. ACM Press.

[310] Wenhua Zhang, Ziyuan Lin, Gansheng Xiao, Junan Chen, Jinshu Wang, and Ying Jiang. LanguageTool proofreading rules evolution and update. In *Proceedings of 2018 International Conference on Big Data Technologies - ICBDT '18*, pages 95–100, New York, New York, USA, 2018. ACM Press.

[311] Zhenchang Xing and E Stroulia. Understanding class evolution in object-oriented software. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 34–43, USA, jun 2004. IEEE.

[312] Zhenchang Xing and Eleni Stroulia. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, dec 2007.

[313] Wujie Zheng, Qirun Zhang, and Michael Lyu. Cross-library API recommendation using web search engines. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, page 480, New York, New York, USA, 2011. ACM Press.

[314] Hao Zhong and Zhendong Su. Detecting API documentation errors. *ACM SIGPLAN Notices*, 48(10):803–816, nov 2013.

[315] Hao Zhong, Suresh Thummalapenta, and Tao Xie. Exposing behavioral differences in cross-language api mapping relations. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 130–145, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[316] Hao Zhong, Suresh Thummalapenta, and Tao Xie. Exposing behavioral differences in cross-language api mapping relations. In *Fundamental Approaches to Software Engineering*, pages 130–145. Springer, Berlin, Heidelberg, 2013.

[317] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, page 195, USA, 2010. ACM Press.

[318] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European*

Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, pages 318–343, Berlin, Heidelberg, 2009. Springer-Verlag.

[319] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. DroidAlarm. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13*, page 353, New York, New York, USA, 2013. ACM Press.

[320] Jing Zhou and Robert J. Walker. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, volume 13-18-Nove, pages 266–277, New York, New York, USA, 2016. ACM Press.

[321] Minhaz F. Zibran, Farjana Z. Eishita, and Chanchal K. Roy. Useful, But Usable? Factors Affecting the Usability of APIs. In *2011 18th Working Conference on Reverse Engineering*, pages 151–155, USA, oct 2011. IEEE.

[322] T. Zimmermann. Card-sorting. *Perspectives on Data Science for Software Engineering*, page 137–141, 2016.

[323] Ilie Şavga, Michael Rudolf, Sebastian Götz, and Uwe Aßmann. Practical refactoring-based framework upgrade. In *Proceedings of the 7th international conference on Generative programming and component engineering*, page 171, USA, 2008. ACM Press.