When APIs are Intentionally Bypassed: An Exploratory Study of API Workarounds

Maxime Lamothe and Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
{max_lam,shang}@encs.concordia.ca

ABSTRACT

Application programming interfaces (APIs) have become ubiquitous in software development. However, external 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 paper 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.

ACM Reference Format:
Maxime Lamothe and Weiyi Shang. 2020. When APIs are Intentionally Bypassed: An Exploratory Study of API Workarounds. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380433

1 INTRODUCTION

As software applications increase in size and complexity, Application Programming Interfaces (APIs) become an integral part of software development [51]. Software is now often produced with help from a slew of APIs to speed up development and reduce project overhead [15, 18]. Dependencies on general purpose frameworks and libraries have been shown to impact a large proportion of client project source code (up to 62%) [7]. Furthermore, publicly available libraries are intricately interconnected. For example, the Maven repository contains over 2.4 million reusable artifacts with over 9 million dependency relationships [8].

This increased development speed comes at a price. By relying on APIs, developers inevitably couple some of their functionality to APIs over which they have little control [9]. This can be a challenge to developers as they are forced to deal with ever-evolving APIs [18, 47]. Difficulties with changing dependencies have led to terminology like "dependency hell" [9]. Studies show that packages are updated within two months of their releases more than 50% of the time [17]; while a sizable portion of API changes break backwards compatibility [68] and have been found to impact 39% of software that uses the APIs [10]. For example, the now infamous "left-pad" incident broke thousands of libraries, including React and Babel due to the removal of 11 lines of code from NPM [22].

More importantly, APIs, as software products themselves, may suffer from typical software issues, such as defects or missing features [13]. However, due to the high cost associated with changing or migrating to a different API [18], developers who use these APIs (API users) ultimately suffer from API defects [9, 39]. Heavy dependence on APIs has become a costly challenge for API users.

Knowledge gaps exist between API developers and the API users [16, 54]. Often, API developers only communicate about their APIs through API documentation such as wikis, manuals, tutorials, or API code examples [14], which are only indirectly linked to the API [16]. On the other hand, 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 [9]. Lacking a direct feedback channel from the API users can lead to situations where API developers must rely on repeated user complaints to become aware of an existing problem [14].

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 [9]. In this paper 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, since these workarounds are created by API users as temporary solutions, many of these workarounds may become technical debt [50], endangering code quality and increasing future maintenance cost [70]. On the other hand, interestingly, these API workarounds may become a vehicle for the API developers to gain

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380433

feedbacks from API users, in order to improve the APIs (e.g., fixing defects in the API).

In this paper, 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 since 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.

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.

Since 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. Therefore, using five open-source APIs, we detected these three patterns of API workarounds 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 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 gap between API developer and API users in order to assist in the development and maintenance of APIs.

Paper Organization. Section 2 provides real examples of API workarounds requested on Stack Overflow to motivate this study. Section 3 contains a qualitative study on API workarounds. Section 4 presents three generalized API workaround patterns extracted from API workaround implementations in Stack Overflow answers. Section 5 presents an empirical study conducted to find and report instances of three API workaround patterns. Section 6 provides a brief summary of related prior work. Section 7 describes threats to the validity of this study. Finally, Section 8 concludes the paper.

2 A MOTIVATING EXAMPLE

In this section, we present an example of API workaround inquiry on Stack Overflow.

Figure 1 presents a question (Stack Overflow id: 34945023) in which the poster (API user) inquires about working around the Roslyn API. In particular, 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

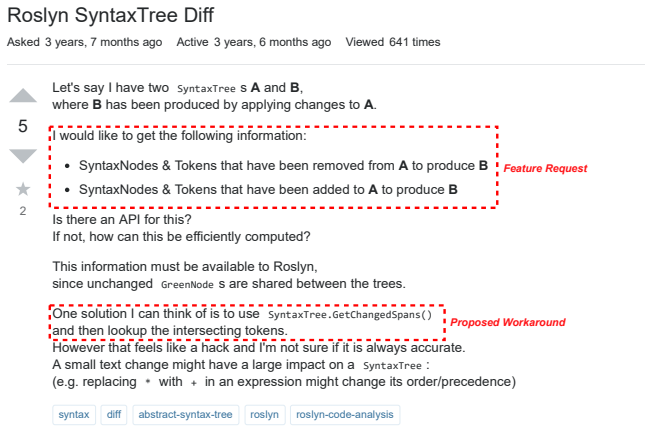

Figure 1: An example of a developer requesting access to data that exists in the Roslyn API but appears unaccessible

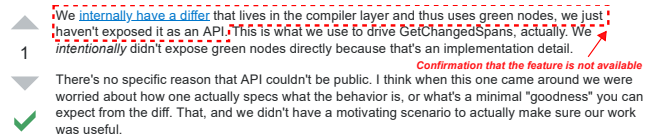

Figure 2: Example of an API developer answering an API workaround request for data that exists in the Roslyn API but appears unaccessible

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 2, was provided by one of the API's developers. Therefore, this post provides rare insight into a direct information exchange between an API user requesting an API workaround, and an API developer.

The API developer confirms that, as suspected by the API user, the feature requested by the user is indeed available internally, but was not exposed to the public since 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. 3) API workarounds provide valuable information for API developers in order to understand the needs from the API users and to improve their APIs.

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 paper we explore API workarounds

and the knowledge that they contain, starting by performing a qualitative study on Stack Overflow posts.

3 INQUIRIES ABOUT API WORKAROUNDS: A QUALITATIVE STUDY

In this section, 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.

3.1 Collecting API workaround related posts

As presented in Section 2, we know that there exist Stack Overflow posts that inquire about API workarounds. Example posts, like the one presented in Figure 1, show that API users can request workarounds to request extra functionality. Meanwhile, prior work has shown that API users can use workarounds to bypass API defects [57]. Therefore, 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. In this section, we therefore seek to systematically determine the various reasons why API users request API workarounds. 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 [1]. 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 [11]. 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.

In order 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) [2] 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. Since 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 consider question and answer pairs as API workaround inquiries.

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.

In order 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 [37] that allows the sorting of posts into categories while also allowing the generation of the categories [55, 74]. 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

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

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 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.

3.3 Measuring coder agreement in our qualitative study

To ensure that the coding derived in Section 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 [6, 32]. Coder agreement is important for trust and reproducibility. Low agreement may lead to non-reproducible results.

We used Krippendorff's α [6, 31, 32] to measure the inter-coder agreement of our qualitative study since it is a general and standard reliability measure [23]. Krippendorff's α can be used to determine a quantitative agreement between coders of typically unstructured data [32].

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 $\alpha \longrightarrow 1$, 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 [31].

Krippendorff's α requires a single value to be assigned to each coded item [32]. 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]

3.4 Qualitative study results

A Krippendorff's $\alpha \geq 0.800$ demonstrates reliable agreement [32]. As shown in Table 1 the Krippendorff's α for our question coding is 0.848, and the Krippendorff's α for our answer coding is 0.810. Therefore, based on Krippendorff's α **the results of our qualitative study are reliable**.

Table 1: Qualitative study reliability coefficient (Krippendorff's α)

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

The coding and categorization process described in Section 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.

3.4.1 *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 2.

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

[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*).

Table 2: Categories of questions derived from Stack Overflow posts on API workarounds

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

issues and the migration between API versions is a well-known problem that has been studied extensively [12, 28–30, 33, 45, 49].

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

3.4.2 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 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 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.

3.4.3 Unusable inquiries.

Due to the nature of the heuristics we used to filter the Stack Overflow post dataset [1], 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).

Table 3: Categories of answers derived from Stack Overflow posts on API workarounds

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

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.

4 PATTERNS FOR IMPLEMENTING API WORKAROUNDS

Section 3 shows that a considerable number of API workarounds 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 3 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 Table 4).

Pattern 1: Functionality extension

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 Table 4 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.

Pattern 2: Deep copy
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 Table 4 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.

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 use the log4j logger context to circumvent exception logging issues experienced by Log4j2. This is presented in the example in Table 4 [3]
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 are attempting to use two or more versions of an API in a given project. In the case of Java, it is not possible to statically load two versions of the same library since 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 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.

[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.

5 REPORTING REAL-LIFE 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.

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 [25] and symbolic link resolver JavaParser [25]. 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.

5.1.1 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.

Table 4: API workaround patterns and corresponding examples

Pattern name	Example
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 executeAndExtract(PreparedStatement insert, SessionImplementor session) throws SQLException {` `session.getTransactionCoordinator().getJdbcCoordinator().getResultSetReturn()` `.executeUpdate(insert);` `ResultSet rs = null;` `try {` `rs = insert.getGeneratedKeys();` `return IdentifierGeneratorHelper.getGeneratedIdentity(rs,` `unquotedIdentifier(persister.getRootTableKeyColumnNames()[0]),` `persister.getIdentifierType()` `);` `} finally {` `if (rs != null) {` `session.getTransactionCoordinator().getJdbcCoordinator().release(rs, insert);` `}` `}` `}` `protected String unquotedIdentifier(String identifier) { //This is a hack.` `if(identifier.startsWith(dialect.openQuote() + "")) {` `return identifier.substring(1, identifier.length() - 1);` `}` `return identifier;` `}` `}`
Deep copy	`@Override` `public String getText(){` `JsonToken t = _currToken;` ... `return _getText2(t);` `}` `protected String _getText2(JsonToken t){` ... `switch (t) {` `case FIELD_NAME:` `return _parsingContext.getCurrentName();` `case VALUE_NUMBER_FLOAT:` `return _textBuffer.contentsAsString();` `}` `return t.asString();` `}` *Keep interface the same, but modify existing information*
Multi-version	`private Set<String> resolveProperties(Map<String, PropertyContext> pCon){` ... `classLoader = getLoader(getStringOrArrayLitteral(pCon.beanValue()));` `URI log4JUri = null;` `if (pCon.containsKey("log4j.configURL")){` `log4JUri = getLitteral(pCon.beanValue(), i->i.stringLiteral().getText(), j->{})` `}` *Url for Log4j config* `if (log4JUri != null) {` `InputStream is = log4JUri.toURL().openStream());` `int toread = 0;` `while ((toread = is.available()) != 0) {is.skip(toread);}` `LoggerContext ctx = (LoggerContext) LogManager.getContext(classLoader, true);` `ctx.setConfigLocation(log4JUri);` `}` *Dynamically set/load Log4j config rather than use current Log4j2 config* ... `}`

The code presented in this table has been edited due to space constraints.

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 [21]. Prior research on 10,000 GitHub projects has shown that Google Guava was the 8th most popular Java library in 2013 [66]. 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 [53]. 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 [24]. 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 [66]. 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 [5]. As of the writing of this paper, over 4,260 maven artifacts have the Apache Log4j Core as a direct dependency [41]. We targeted 20 different versions of the Log4j API, from version 2.0.1 to 2.11.2.

5.1.2 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 5.1.1.

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.

5.1.3 Detecting patterns.

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

Since 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 [61], and reduce the pattern detection time. Furthermore, since JavaParser [25] 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 [61]. As a first detection step, we leverage JavaParser [25] 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 [25] to obtain code bindings in user projects.

Functionality extension: To detect the *Functionality extension* pattern, we use the binding information obtained through JavaParser [25] 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 [25] 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

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

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.

5.2 Results and discussion

In this section 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 are 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.

5.2.1 *Functionality extension:*

During our manual observation we were able to extract nine *Functionality extension* API workarounds from the selected user systems. By searching through forums and patch notes, we were able to 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 were able to 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.

5.2.2 *Deep copy:*

We were able to 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.

5.2.3 *Multi-version:*

We were able to 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 were able to 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.

5.2.4 *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 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 RELATED WORK

In this section we discuss related research in the field of APIs. We specifically concentrate on prior work based on API usage patterns, API misuse, and using Stack Overflow as an API resource.

API usage. API usage patterns have been used in prior work to understand how APIs evolve [38], to detect API compatibility issues [57], for API migration [27, 33, 43], code recommendation [42, 44] and more. In this paper we specifically concentrate on usage that would not typically be expected from API developers (i.e. workarounds). API usage patterns can be extracted from source code [27, 33, 38, 42–44, 57], but they can also be extracted from code examples [19, 71]. In this paper we use a mixture of both approaches by extracting and generalizing the usage of three API workaround patterns from code examples taken from Stack Overflow to later detect instance of these patterns in source code.

API misuse. API misuses can be characterized by their violation of an APIs constraints [4]. Misusing APIs can lead to an increase in software security vulnerabilities and bugs [3, 20, 40]. Prior work has concentrated on identifying and detecting API misuses [4, 34, 36, 40, 46, 59, 65, 67]. API misuse detection tools often rely on frequent usage patterns to detect API uses that deviate from the norm, but they can also rely on Mutation analysis [67].

The API workaround inquiries and patterns deliberated in this paper are instances of unconventional API usage. API users sometimes seek to modify or work around the current implementation of an API. This paper concentrates on constructive workarounds that could be seen as improvements to the API. However, there exist non-constructive ways to use APIs. These usage patterns are known as API misuses. API misuses can be considered as unintentional alternate uses of APIs, meanwhile API workarounds are intentional alternate uses of APIs.

Stack Overflow as an API resource. Stack Overflow has been used as a source of information for several prior works on APIs [19, 35, 64, 71]. Stack Overflow has been used for API topics such as how API changes trigger discussions [35, 63], improving API documentation [47, 58, 60], understanding the concerns of developers [62], API recommendation [48, 52], indexing API information [69], and API deprecation [73].

However, Stack Overflow is not the only online community that has been used as a source of data for API research. Other forums have also been used to develop tools that could find negative developer sentiments and highlight, search, and estimate API "hot topics" [72]. Online forums have also been used to develop automatic code critics that can inform API users of API usage rules and support API usage [56]. We similarly use online information from Stack Overflow to understand how developers use APIs. However, our work differs in scope since we wish to understand how and why API users seek to work around missing or problematic API functionality. We then use this information to generate API workaround patterns that can be used by API developers to search for new and desired features to add to their API.

7 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 paper do not generalize to other programming languages. However, while the strategies presented in this paper 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 paper might not be fully indicative of API workarounds and instead reflect internal bias. We attempted to mitigate these threats by having the authors of this paper 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.

8 CONCLUSION

In this paper we conduct 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. Our paper makes the following contributions:

(1) To the best of our knowledge, 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.

REFERENCES

[1] [n.d.]. Stack Exchange Data Dump : Stack Exchange, Inc. : Free Download, Borrow, and Streaming. https://archive.org/details/stackexchange

[2] 2019. https://www.nltk.org/

[3] S Amann, S Nadi, H A Nguyen, T N Nguyen, and M Mezini. 2016. MUBench: A Benchmark for API-Misuse Detectors. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 464–467.

[4] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating Next Steps in Static API misuse Detection. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR 19)*. IEEE Press, Piscataway, NJ, USA, 265–275.

[5] Apache. 2019. Apache/Log4j. https://github.com/apache/logging-log4j2

[6] Ron Artstein and Massimo Poesio. 2008. Inter-Coder Agreement for Computational Linguistics. *Computational Linguistics* 34, 4 (2008), 555–596.

[7] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache Community Upgrades Dependencies: An Evolutionary Study. *Empirical Softw. Engg.* 20, 5 (Oct. 2015), 1275–1317.

[8] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The Maven Dependency Graph: A Temporal Graph-based Representation of Maven Central. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, Piscataway, NJ, USA, 344–348.

[9] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. 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)*. ACM, New York, NY, USA, 109–120.

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

[11] William B. Cavnar and John M. Trenkle. 1994. N-Gram-Based Text Categorization. In *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*. 161–175.

[12] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 55, 11 pages.

[13] Barthélémy Dagenais and Martin P. Robillard. 2009. SemDiff: Analysis and recommendation support for API evolution. *Proceedings - International Conference on Software Engineering* (2009), 599–602.

[14] Barthélémy Dagenais and Martin P. Robillard. 2010. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 127–136.

[15] Barthélémy Dagenais and Martin P. Robillard. 2011. Recommending Adaptive Changes for Framework Evolution. *ACM Transactions on Software Engineering and Methodology* 20, 4 (2011), 1–35.

[16] Barthelemy Dagenais and Martin P. Robillard. 2012. Recovering traceability links between an API and its learning resources. *2012 34th International Conference on Software Engineering (ICSE)* (2012).

[17] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (01 Feb 2019), 381–416.

[18] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (2006), 83–107.

[19] Glassman E., Zhang T., Hartmann B., and Kim M. 2018. Visualizing API Usage Examples at Scale. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI 2018* (2018).

[20] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 73–84.

[21] Google. 2019. google/guava. https://github.com/google/guava

[22] David Haney. 2016. NPM and left-pad: Have We Forgotten How To Program. https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program

[23] Andrew F. Hayes and Klaus Krippendorff. 2007. Answering the Call for a Standard Reliability Measure for Coding Data. *Communication Methods and Measures* 1, 1 (2007), 77–89.

[24] Jackson. 2019. FasterXML/Jackson. https://github.com/FasterXML/jackson

[25] JavaParser. 2019. JavaParser. https://javaparser.org/

[26] JUnit. 2019. Junit. https://junit.org/junit4

[27] Miryung Kim. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. *ICSE* (2013), 502–511.

[28] Miryung Kim, David Notkin, and Dan Grossman. 2007. Automatic inference of structural changes for matching across program versions. *Proceedings - International Conference on Software Engineering* (2007), 333–342.

[29] Miryung Kim, David Notkin, and Dan Grossman. 2007. Automatic Inference of Structural Changes for Matching Across Program Versions. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 333–343.

[30] Sunghun Kim, Kai Pan, and E. James Whitehead. 2005. When functions change their names: Automatic detection of origin relationships. *Proceedings - Working Conference on Reverse Engineering, WCRE* 2005 (2005), 143–154.

[31] Klaus Krippendorff. 2011. Computing Krippendorffs Alpha Reliability. *University of Pennsylvania Scholarly Commons* (Jan 2011).

[32] Klaus H. Krippendorff. 2013. *Content Analysis - 3rd Edition: an Introduction to Its Methodology*. SAGE Publications, Inc.

[33] M. Lamothe and W. Shang. 2018. Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android. *MSR '18: 15th International Conference on Mining Software Repositories* (2018).

[34] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 306–315.

[35] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How do API changes trigger stack overflow discussions? a study on the Android SDK. *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014* (2014).

[36] Christian Lindig. 2015. Chapter 2 - Mining Patterns and Violations Using Concept Analysis. In *The Art and Science of Analyzing Software Data*, Christian Bird, Tim Menzies, and Thomas Zimmermann (Eds.). Morgan Kaufmann, Boston, 17 – 38.

[37] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. 2015. 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)*. ACM, New York, NY, USA, 415–425.

[38] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of API stability and adoption in the android ecosystem. *IEEE International Conference on Software Maintenance, ICSM* (2013), 70–79.

[39] Tyler Mcdonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. *2013 IEEE International Conference on Software Maintenance* (2013).

[40] Martin Monperrus and Mira Mezini. 2013. Detecting Missing Method Calls As Violations of the Majority Rule. *ACM Trans. Softw. Eng. Methodol.* 22, 1, Article 7 (March 2013), 25 pages.

[41] MVN. 2019. Apache/Log4j. https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core

[42] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016* i (2016), 511–522.

[43] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. 2010. A Graph-based Approach to API Usage Adaptation. *SIGPLAN Not.* 45, 10 (oct 2010), 302–321.

[44] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1050–1060.

[45] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping API Elements for Code Migration with Vector Representations. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 756–758.

[46] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. 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)*. ACM, New York, NY, USA, 383–392.

[47] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. 2012. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. *Georgia Tech Technical Report* (2012), 1–11.

[48] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. 2018. 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)*. ACM, New York, NY, USA, 632–642.

[49] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen. 2017. Statistical Migration of API Usages. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 47–50.

[50] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. 91–100.

[51] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. *IEEE International Conference on Software Maintenance, ICSM* (2012), 378–387.

[52] M. M. Rahman, C. K. Roy, and D. Lo. 2016. RACK: Automatic API Recommendation Using Crowdsourced Knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 349–359.

[53] RedHat. 2019. Hibernate OGM. http://hibernate.org/orm

[54] Romain Robbes and Mircea Lungu. 2011. A study of ripple effects in software ecosystems: (NIER track). *2011 33rd International Conference on Software Engineering (ICSE)* (2011), 904–907.

[55] Gordon Rugg and Peter Mcgeorge. 1997. The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts. *Expert Systems* 14, 2 (1997), 80–93.

[56] C R Rupakheti and D Hou. 2012. Evaluating forum discussions to inform the design of an API critic. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 53–62.

[57] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, Piscataway, NJ, USA, 288–298.

[58] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* (2014), 643–652.

[59] S. Thummalapenta and T. Xie. 2009. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 283–294.

[60] Christoph Treude and Martin P. Robillard. 2016. Augmenting API documentation with insights from stack overflow. *Proceedings of the 38th International Conference on Software Engineering - ICSE 16* (2016).

[61] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2016), e1838.

[62] Pradeep K. Venkatesh, Shaohua Wang, Feng Zhang, Ying Zou, and Ahmed E. Hassan. 2016. What Do Client Developers Concern When Using Web APIs? An Empirical Study on Developer Forums and Stack Overflow. *2016 IEEE International Conference on Web Services (ICWS)* (2016).

[63] Shaohua Wang, Iman Keivanloo, and Ying Zou. 2014. How Do Developers React to RESTful API Evolution ? *12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings* 8831 (2014), 245–259.

[64] Shaohua Wang, NhatHai Phan, Yan Wang, and Yong Zhao. 2019. Extracting API Tips from Developer Question and Answer Websites. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, Piscataway, NJ, USA, 321–332.

[65] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 35–44.

[66] Tal Weiss. 2019. We Analyzed 30,000 GitHub Projects - Here Are The Top 100 Libraries in Java, JS and Ruby. https://blog.overops.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/

[67] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exposing Library API Misuses via Mutation Analysis. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 866–877.

[68] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering* Dcc (2017), 138–147.

[69] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. 2017. Learning to extract API mentions from informal natural language discussions. *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* (2017), 389–399.

[70] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the Impact of Design Debt on Software Quality. In *Proceedings of the 2Nd Workshop on Managing Technical Debt (MTD '11)*. ACM, New York, NY, USA, 17–23.

[71] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and Supporting Adaptation of Online Code Examples. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 316–327.

[72] Y Zhang and D Hou. 2013. Extracting problematic API features from forum discussions. In *2013 21st International Conference on Program Comprehension (ICPC)*. 142–151.

[73] Jing Zhou and Robert J. Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016* (2016), 266–277.

[74] T. Zimmermann. 2016. Card-sorting. *Perspectives on Data Science for Software Engineering* (2016), 137–141.