Historical and Impact Analysis of API Breaking Changes: A Large-Scale Study

Laerte Xavier, Aline Brito, Andre Hora, Marco Tulio Valente
ASERG Group
Department of Computer Science (DCC)
Federal University of Minas Gerais, Brazil
{laertexavier,alinebrito,hora,mtov}@dcc.ufmg.br

***Abstract*—Change is a routine in software development. Like any system, libraries also evolve over time. As a consequence, clients are compelled to update and, thus, benefit from the available API improvements. However, some of these API changes may break contracts previously established, resulting in compilation errors and behavioral changes. In this paper, we study a set of questions regarding API breaking changes. Our goal is to measure the amount of breaking changes on real-world libraries and its impact on clients at a large-scale level. We assess (i) the frequency of breaking changes, (ii) the behavior of these changes over time, (iii) the impact on clients, and (iv) the characteristics of libraries with high frequency of breaking changes. Our large-scale analysis on 317 real-world Java libraries, 9K releases, and 260K client applications shows that (i) 14.78% of the API changes break compatibility with previous versions, (ii) the frequency of breaking changes increases over time, (iii) 2.54% of their clients are impacted, and (iv) systems with higher frequency of breaking changes are larger, more popular, and more active. Based on these results, we provide a set of lessons to better support library and client developers in their maintenance tasks.**

***Index Terms*—Software Evolution; API Usage; API Stability; Backwards Compatibility.**

I. Introduction

In software development, change often occurs to accommodate new features, fix bugs, and refactor source code. Software libraries, commonly used nowadays to improve development productivity and support functionality reuse on client applications [1], [2], are not different and also evolve over time [3]. These functionalities are provided by libraries to clients via *Application Programming Interfaces* (APIs), which are contracts that client applications rely on [4]. Ideally, APIs should be *backward-compatible* when evolving, *i.e.,* do not break contracts with their client applications.

In practice, breaking client applications is a common practice: previous studies indicate that APIs are usually *backward-incompatible* [5]–[8]. In this context, several solutions have been proposed to mitigate the impact on clients (*e.g.,* [9]–[13]). For example, by mining version history, some studies suggest how client applications should be updated due to broken API elements (*e.g.,* a public method removed from an old library version). However, even though there are solutions to alleviate the impact of library evolution, we are still unaware about the real size of this impact on client applications: to what extent are clients affected by backward-incompatibility? Furthermore, we are unsure whether backward-incompatibility tends to get better (or worse) over time: is this a problem only faced by newer (and possibly "unstable") libraries or older (and "stable") ones should also take special care with API compatibility?

In this paper, we study a set of questions regarding API breaking changes. We analyze (i) the frequency of API breaking changes, (ii) the behavior of these changes over time, (iii) the impact on client applications, and (iv) the characteristics of libraries with high frequency of breaking changes. Our main goal is twofold: to measure the amount of breaking changes on real world libraries and its impact on clients at a large-scale level. Therefore, we investigate the following research questions to support our study:

- *RQ1.* What is the frequency of API breaking changes?
- *RQ2.* How do API breaking changes evolve over time?
- *RQ3.* What is the impact of API breaking changes in client applications?
- *RQ4.* What are the characteristics of libraries with high and low frequency of breaking changes?

In this study, we analyze 317 real-world Java libraries, 9K releases, and 260K client applications. Our results show that (i) 14.78% of the API changes break compatibility with previous versions, (ii) the frequency of breaking changes increases over time, (iii) only 2.54% of their clients are potentially impacted, and (iv) libraries with higher frequency of breaking changes are larger, more popular, and more active. Based on these results, we provide a set of lessons to better support library and client developers in their maintenance tasks. Therefore, the contributions of this paper are summarized as follows:

- We provide a large-scale study to better understand the extension and the impact of API breaking changes.
- We provide lessons learned from our API analysis to support library/client developers in maintenance activities

Structure of the paper: Section II presents the background on API changes. We describe our experiment design in Section III and present the experiment results in Section IV. Summary and findings are described in Section V. Section VI states threats to validity and Section VII presents related work. Finally, we conclude the paper in Section VIII.

II. Background

Libraries provide interfaces to software components created to be reused by client applications [4]. They take advantage of visibility modifiers to expose *interfaces* meant to be stable (*e.g.,* in Java they use `public` and `protected` modifiers). As any other software system, during their life cycle, libraries are also subjected to evolutionary changes, such as addition, removal, or modification of their API elements, including types, fields, and methods. However, the cost of evolving libraries may become higher due to the impact on external clients. In this context, API changes are classified into *breaking changes* and *non-breaking changes* [14], as follows:

- *Breaking changes.* Changes that break backward compatibility through removal or modification of API elements. As a consequence, clients may face compilation errors after updating.
- *Non-breaking changes.* Changes that preserve compatibility and usually involve addition of new functionalities to the library. Thus, migrating between API versions including only non-breaking changes does not cause negative effects to client applications.

In this work, we define and implement an API *diff* tool to identify *breaking changes* and *non-breaking changes* between two versions of a Java library. Table I details the changes evaluated in our *diff*. In the case of types, breaking changes include removal of a type, change on its visibility modifier (*e.g.,* from public to protected), and change in the type's supertype. Breaking changes in fields include, for example, changes in the field's type or default value. Breaking changes in methods include, for example, changes in their signatures. By contrast, non-breaking changes include addition of new elements and change on visibility modifiers (*e.g.,* from private to public or protected). Furthermore, changes in deprecated elements (*e.g.,* deprecated method removal) are classified as *non-breaking changes* by our *diff* tool, because developers in this case have been previously alerted about the risks of using deprecated elements. A similar decision is followed in other studies about breaking changes [14].

TABLE I
EVALUATED BREAKING AND NON-BREAKING CHANGES.

Category	API Element	List of Changes
Breaking	Type	REMOVAL, VISIBILITY LOSS, SUPERTYPE CHANGE
	Field	REMOVAL, VISIBILITY LOSS, TYPE CHANGE, DEFAULT VALUE CHANGE
	Method	REMOVAL, VISIBILITY LOSS, RETURN TYPE CHANGE, PARAMETER LIST CHANGE, EXCEPTION LIST CHANGE
Non-breaking	All	ADDITION, VISIBILITY GAIN, DEPRECATION OPERATION

III. Study Design

A. Selecting Java Libraries

To answer the proposed research questions, we analyzed the most popular Java libraries hosted on GitHub. First, we selected the top 1,000 repositories ordered by number of stars. Then, we manually classified them into *library* (554 repositories, 55.40%) and *non-library* (446 repositories, 44.60%). This manual classification was performed by the first author of this paper with the support of each repository web page and documentation. Finally, from the *library* group, we discarded the ones in the first quartile of number of releases and age, as follows:

- *Number of releases.* We selected libraries with two or more releases (*i.e.,* first quartile equals to 1). We applied this criteria to focus on active libraries and to ensure at least one pair of releases to be compared in each library.
- *Age.* We selected systems with more than 515 days from the first commit (*i.e.,* first quartile equals to 515 days). We use this criteria to assess libraries with a relevant evolution and to ensure historical data to our analysis.

Based on this filtering criteria, the final selection has *317 libraries*[1], including well-known ones such as FACEBOOK/REACTNATIVE, GOOGLE/GUAVA, and JUNIT-TEAM/JUNIT4. To better characterize these libraries, Figure 1 presents the distribution of number of releases and age (in years) as well as number of stars and files. We provide violin plots for *all* 1,000 initial repositories, for the 554 repositories categorized as *library*, and for the 317 *studied* libraries. Violin plots are useful for presenting the distribution of data because besides embedding a box plot they also show the probability density of the data at different values. Considering the *studied* libraries, we have the following results. For number of releases, the first quartile, median, and third quartile are 6, 15, and 29 releases. For age, the quartiles are 2.2, 3.4, and 5.2 years. For number of stars, the first quartile, median, and third quartile are 1,216, 1,792 and 3,215 stars. Finally, for number of files, the quartiles are 1,298, 6,676, and 25,335 files. We observe that the *studied* libraries are statistically significant different for number of releases, age, and number of stars (*p-value* < 0.05 for Mann-Whitney test), when compared to the *all* repository and also to the *libraries* repository. However, they are not statistically different for number of files.

B. Extracting API Breaking Changes (RQ1 and RQ2)

To measure the frequency of API breaking changes, we first identified all API changes as described in Section II. We implemented a parser based on the Eclipse JDT library to compute the *diff* between API elements (types, fields, and methods) from two library releases. We focused on public and protected API elements because they represent the external contract between libraries and clients. Let R_n be the last release and R_1 the first one of a given library. To answer *RQ1*, we computed the *diff* between releases R_n and R_{n-1},

[1]Further description of repositories labeled as *All*, *Library*, and *Studied* available at: https://goo.gl/BU4W0C

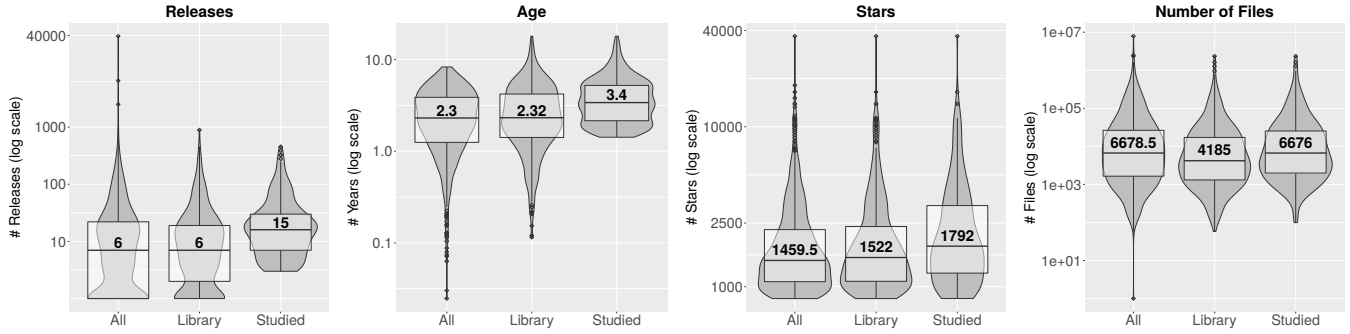

Fig. 1. Repositories distributions by releases, age, stars, and files.

i.e., diff(R_n,R_{n-1}). To answer *RQ2*, we compared *all* releases (from R_1 to R_n) of a library, *i.e., diff(R_i,R_{i-1})* for $i = 2 \ldots n$. To better understand this data in *RQ2*, we summarized the breaking changes over time per year.

C. Measuring API Breaking Changes Impact (RQ3)

To support answering *RQ3*, we calculated the impact of the breaking changes identified in *RQ1* on client systems. Using an ultra-large dataset of Java systems, we counted the ones that feature an *import* statement to the detected breaking changes. For types, we perform a direct analysis by looking for their qualified names. For methods and fields, on the other hand, we assess the *imports* of their enclosing type. In other words, if a breaking change is detected in a method `m` of a class `C`, we count as potentially impacted all clients that import `C`. This approach at least retrieves the worst case scenario of the potential impact measure.

To collect the client systems, we used JAVALI[2], a tool to measure popularity of Java libraries. This tool works on the top of a dataset with more than 260K Java systems and it is based on the BOA infrastructure [15], [16]. As an example, Table II shows the top-5 most used types, with their corresponding number of clients. The number of clients range from 88K (`java.io.file`) to 143K (`java.util.ArrayList`).

TABLE II
TOP-5 POPULAR TYPES.

Position	Name	Number of Clients
1	`java.util.ArrayList`	143,454
2	`java.io.IOException`	136,058
3	`java.util.List`	134,053
4	`java.util.HashMap`	94,220
5	`java.io.File`	88,703

D. Comparing Libraries with High and Low Frequency of Breaking Changes (RQ4)

In order to distinguish libraries with low and high rates of API breaking changes, we classified the studied libraries in two groups (*top* and *bottom*) to answer *RQ4*. We then collected a set of project metrics (such as activity, size, etc) to compare both groups. The goal is to verify whether these metrics have an impact on the number of API breaking changes. This process is summarized in the following three steps.

1. Defining metrics likely to impact breaking changes. To analyze libraries with high and low rate of breaking changes, we define five dimensions related to open source development and social coding: popularity, size, community, activity, and maturity. For each of them, we define specific metrics to measure and characterize the studied libraries. These metrics were also used in a previous study about the adoption of replacement messages in API deprecation [8]. Each dimension and the corresponding metrics are described as:

- *Popularity.* Represents how popular a library is on GitHub. The metrics are *number of stars*, *number of watchers*, and *number of forks*.
- *Size.* Characterizes the library volume of artifacts. The associated metrics are *number of files* and *number of API elements* (*i.e.,* sum of public and protected types, fields, and methods).
- *Community.* Represents the library community size. The metrics are *number of contributors*, *average files per contributor*, and *average API elements per contributor*.
- *Activity.* Characterizes the activity level of a library development team. The metrics are *number of commits*, *number of releases*, and *average days per release*.
- *Maturity.* Represents the age of a library. The associated metric is *number of years*.

2. Selecting Top and Bottom libraries. We consider two groups of libraries: the ones with low rate of breaking changes, labeled as *top libraries*, and a second group, labeled as *bottom libraries*, with higher rates of breaking changes.

We first identified the active libraries, *i.e.,* the ones with at least one API change (either breaking or non-breaking), resulting in 235 libraries. Then, we sorted these 235 libraries, in ascending order, by the percentage of API breaking changes. Finally, we ended up with two groups: top-25% (*i.e.,* libraries with the lowest percentage of breaking changes) and bottom-25% (*i.e.,* libraries with the highest percentage); each group with 58 libraries. Figure 2 shows the distribution of breaking changes in each group. As expected, the median percentage of

[2]http://java.labsoft.dcc.ufmg.br/javali

changes is low (0%) for *top libraries* and very high (73.75%) for *bottom* ones. Table III shows the name of five top and five bottom libraries. All top libraries in this table have no breaking changes at all; by contrast, in the bottom libraries, all detected API changes are classified as breaking changes.

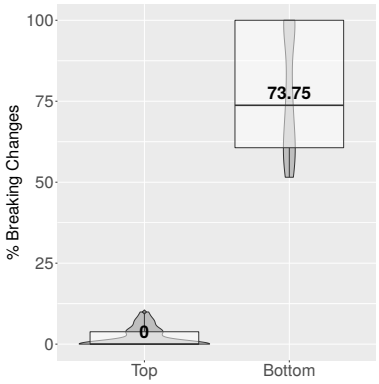

Fig. 2. Breaking changes distribution in top-25% and bottom-25% libraries.

TABLE III
EXAMPLE OF *Top* AND *Bottom* LIBRARIES ORDER BY NUMBER OF BREAKING CHANGES.

Group	Library	Breaking Changes
Top	CHRISBANES/ACTIONBAR-PULLTOREFRESH	0 (0%)
	GOOGLEMAPS/ANDROID-MAPS-UTILS	0 (0%)
	FACEBOOK/CONCEAL	0 (0%)
	CHRISJENX/CALLIGRAPHY	0 (0%)
	MIKEPENZ/ABOUTLIBRARIES	0 (0%)
Bottom	KYMJS/KJFRAMEFORANDROID	4 (100%)
	GRAILS/GRAILS-CORE	5 (100%)
	MONGODB/MONGO-HADOOP	5 (100%)
	LIAOHUQIU/CUBE-SDK	19 (100%)
	ZEROMQ/JEROMQ	23 (100%)

3. Extracting metrics and comparing libraries. We extracted the metrics for both *top* and *bottom* libraries and then compared the obtained values. First, we analyze the statistical significance of the difference between both groups by applying the Mann-Whitney test at *p-value* = 0.05. To show the effect size of the difference between them, we compute Cliff's Delta (or *d*). Following the guidelines in [17]–[19], we interpret the effect size values as small for $0.147 < d < 0.33$, medium for $0.33 < d < 0.474$, and large for $d < 0.474$.

IV. RESULTS

RQ1: What is the frequency of API breaking changes?

We analyze the frequency of changes for types, fields, and methods between the two latest releases, *i.e., diff*(R_n,R_{n-1}), of the 317 studied libraries. We identified at least one change in 235 libraries (74.13%). From this total, 198 libraries (62.46%) have at least one breaking change, while 218 (92.77%) have at least one non-breaking change. Table IV presents the number of changes per API element (*i.e.,* types, fields, and methods). Considering all of them, 501,645 changes were identified, from which 27.99% are breaking changes and 72.01% are non-breaking changes. Methods are the API elements with more changes, including breaking changes. Considering the 140,460 breaking changes, 27.81% are in methods.

TABLE IV
NUMBER OF API BREAKING AND NON-BREAKING CHANGES.

Element	Total	Breaking Change	Non-Breaking Change
Types	61,897	11,712 (18.92%)	50,185 (81.08%)
Fields	66,953	25,044 (37.41%)	41,909 (62.59%)
Methods	372,795	103,704 (27.81%)	269,091 (72.19%)
All	501,645	140,460 (27.99%)	361,185 (72.01%)

To understand the stability of the studied libraries, Figure 3 presents the distribution of absolute and relative breaking changes. A logarithmic scale is applied to absolute plots so we can better visualize outlier libraries.

Absolute analysis. Figure 3(a) shows the absolute distribution of the number of changes (breaking and non-breaking) per library. Considering all API elements, the first quartile is 0, the median is 22, and the third quartile is 285 changes. On the median, types and fields have two changes while methods have 17. The third quartile for types, fields, and methods is 29, 27, and 206 changes, respectively.

Figure 3(b) details the previous analysis by exploring the absolute distribution of *breaking changes* per library. Considering all API elements, the first quartile is 0, the median is 4, and the third quartile is 75. In absolute terms, we note that types and fields present similar distributions (median equal to 0). However, outlier values are very different: we observe a library with 11,816 breaking changes for fields, and another one with 1,392 breaking changes for types. We manually analyzed both cases. The first one happened in the Android library MANUELPEINADO/FADINGACTIONBAR, when the project structure faced a major change, as described in the commit message: *"Changed project structure so that all subprojects are in the same root."*[3] The second one happened in the graph library NEO4J/NEO4J, when several changes were inserted to improve its design. One example is found in the pull request that removed the type `SchemaRuleContent`: *"This PR makes sure that all types of schema rules are properly checked and simplifies duplicates checking by removal of SchemaRuleContent."*[4]

Relative analysis. Figure 3(c) presents the distribution of the relative number of breaking changes per library. For all API elements, the first quartile is 0%, the median is 14.78%, and the third quartile is 43.35%. Moreover, we found 17 libraries (5.35%) with 100% of breaking changes, such as NETFLIX/ASTYANAX, NATHANMARZ/STORM, and GRAILS/GRAILS-CORE. But in all these cases, the absolute number of changes is also small (at most 23 changes in NETFLIX/ASTYANAX).

[3]More details at: https://goo.gl/VHwzKI
[4]More details at: https://goo.gl/iUzTE8

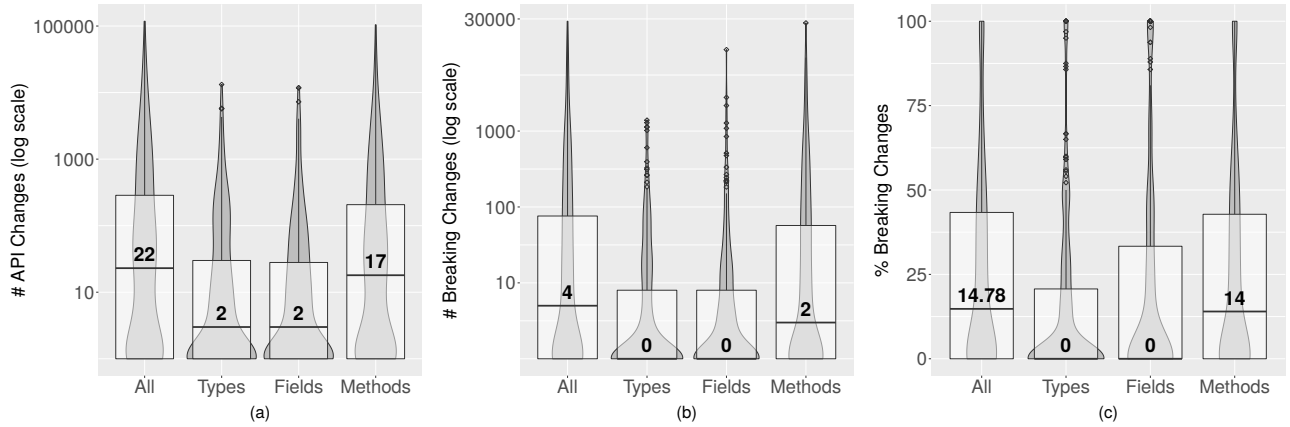

Fig. 3. Distribution of API changes for all elements, types, fields, and methods. (a) Absolute number of all changes, (b) absolute number of breaking changes, and (c) relative number of breaking changes.

Summary: From the 501,645 analyzed API changes, we observe a high rate of breaking changes (27.99%). On the median, 14.78% of the API changes in a library break contracts with clients; the higher ratio of breaking changes occurs on methods.

RQ2: How do API breaking changes evolve over time?

To answer this second research question, we verify *all* releases (from R_1 to R_n) of the 317 studied libraries. The goal is to analyze the frequency of breaking changes over time and, thus, to investigate the impact of software evolution on library stability. To accomplish that, we verify *9,329 releases* and summarize the frequency of breaking changes per year. Because the third quartile of the studied libraries age is 5.2 years, we decided to analyze at most five years of their evolution. In addition, due to our selection criteria discussed in Section III-A, the studied libraries have at least one year.

Figure 4 presents the relative distribution of the means of breaking changes per library and per year. Those with no versions released in a given year were discarded. For each library, we calculate the mean number of breaking changes in each year, by considering only the releases in the year. In this way, we generate distributions per library and per year. In the first year of existence, 232 libraries released public versions. The first quartile of the means is 16.65%; the median, 29.02%; and the third quartile, 42.74%.

For breaking changes in releases during the second year, the first quartile is 15.32%, the median is 31.46%, and the third quartile is 47.72%. From the total, 212 libraries registered at least one release during their second year. From the first to the second year, we observe a light increase of 2.44% in the median value. However, the Mann-Whitney test reveals no statistical significant difference between both groups.

In the third year, 149 libraries were analyzed. The first quartile, the median, and the third quartile are, respectively: 14.73%, 37.12%, and 50.75%. Comparing to the previous years, the median is slightly higher, increasing 5.66% and 8.10% when compared to the second and first years, respectively. Despite of that, the Mann-Whitney test does not show a statistical significant difference between the three years.

In our dataset, 106 libraries have version released during their fourth year. The quartile values are 25.33%, 45.16%, and 59.76%, respectively. The frequency of breaking changes increases by 8.04%, 13.70%, and 16.14% when compared to the third, second, and first years, respectively. In this case, the Mann-Whitney test reveals that this group is statistically significant different from the three previous ones.

Finally, in the fifth year, 83 libraries have released versions. The first quartile is 30.53%, the median is 49.14%, and the third quartile is 62.80%. From the fifth to the fourth years, we do not observe statistical significant difference.

Therefore, the historical analysis of the breaking changes frequency reveal that they increase by 20% in five years (median values). This may be explained by the fact that as time passes, libraries tend to provide API elements that are harder to manage and more likely to change.

Summary: The frequency of breaking changes increases over time. Comparing the first to the fifth year, this number increases by 20% (from 29.02% to 49.14%). This may occur because as libraries evolve, they become larger and more likely to change.

RQ3: What is the impact of API breaking changes in client applications?

In this third research question, we investigate the impact of the breaking changes reported in *RQ1* on client applications. For that, we analyze both the types with breaking changes and the types declaring fields and methods with breaking changes. The goal is to assess the *potential impact* of breaking changes by analyzing *import* statements in client systems. As detailed in Section III-C, our dataset of client applications has around 260K Java systems.

Considering all API elements, 140,460 breaking changes were detected (see Table IV), referring to 16,291 types. From such types, 1,290 (7.91%) potentially impacted at least one client application, *i.e.,* clients with at least one *import* to

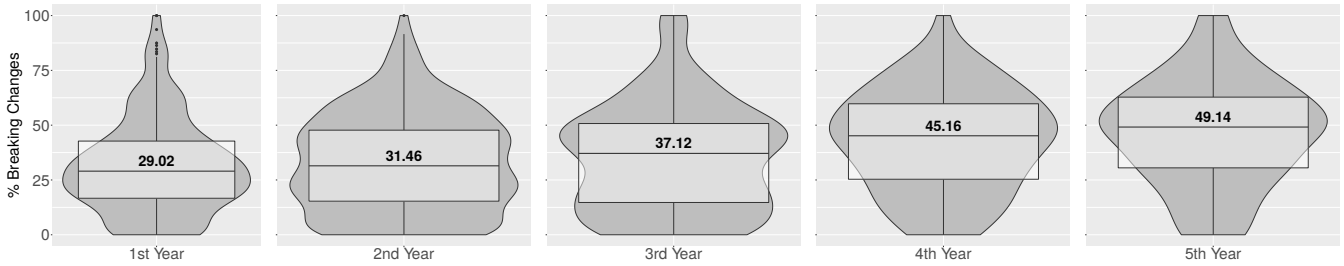

Fig. 4. Distribution of API breaking changes per year. The distribution values are the mean rate of changes in a year, considering the releases produced in this year.

these types in our dataset. For the remaining types with breaking changes, we did not find clients in the JAVALI/BOA dataset; therefore, they were discarded. Figure 5 presents the distribution of absolute and relative number of impacted clients per library/type. A logarithmic scale is applied to absolute plots to ensure outliers visualization.

Figure 5(a) presents the absolute distribution of the number of impacted clients per library. The first quartile is equal to 75 clients; the median, 349; and the third quartile is 2,245 clients. In this context, the top-3 libraries with more clients are JUNIT-TEAM/JUNIT4, with 54,217 clients; SPRING-PROJECTS/SPRING-FRAMEWORK, with 23,793 clients; and GOOGLE/GUAVA, with 12,524 clients.

Figure 5(b) shows the absolute distribution of the number of clients impacted per type with breaking change. The first quartile, the median, and the third quartile are 10, 26 and 90.75 clients, respectively. Despite the low numbers registered by the quartiles, the top-3 types with more impacted clients belong to well-known libraries: `org.junit.Assert` (imported on 10,857 clients), `junit.framework.Assert` (imported on 10,535 clients), and `org.bukkit.plugin.java.JavaPlugin` (imported on 8,005 clients).

Finally, Figure 5(c) details the relative distribution of the impacted clients, *i.e.,* number of clients impacted by a breaking change in a given API divided by the total number of clients of this API. The first quartile is 1%, the median is 2.54%, and the third quartile is 13.10%. The top-3 types with higher rates of breaking change impact are: `*.streaming.video.VideoQuality`, from FYHERTZ/LIBSTREAMING, with 100%; `*.chronicle.Excerpt`, from PETER-LAWREY/JAVACHRONICLE, also with 100%; and `*.scene2d.ui.Label`, from LIBGDX/LIBGDX, with 99.64%. However, in all these three cases, the number of clients of each type is also small (at most 280 clients in LIBGDX/LIBGDX).

Therefore, the impact of breaking changes in terms of impacted clients tends to be low (2.54%, on the median). This may indicate that (i) library developers are careful before inserting breaking changes on used types, or (ii) the changed types are for internal usage only [20], [21]. However, we also notice many outliers (123 types) with more than 32.03% of clients impact. Table V lists the breaking changes with the highest impact on clients.

TABLE V
TOP-10 BREAKING CHANGES WITH THE HIGHEST IMPACT ON CLIENTS.

Type	Impact
`*.streaming.video.VideoQuality`	100.00%
`*.chronicle.Excerpt`	100.00%
`com.badlogic.gdx.scenes.scene2d.ui.Label`	99.64%
`android.content.res.AssetManager`	97.87%
`*.mustachejava.DefaultMustacheFactory`	96.47%
`android.telephony.TelephonyManager`	95.86%
`com.android.volley.RequestQueue`	93.52%
`org.bukkit.plugin.java.JavaPlugin`	93.45%
`org.pegdown.PegDownProcessor`	91.67%
`android.widget.AbsListView`	90.51%

Summary: 2.54% of the client applications are potentially impacted by breaking changes, on the median. One possible explanation is that developers may be careful to break widely used types.

RQ4: What are the characteristics of libraries with high and low frequency of breaking changes?

To analyze libraries with high and low percentage of breaking changes, we compare *top* and *bottom* libraries as described in Section III-D. The purpose is to verify whether library popularity, size, community, activity, and maturity impact the frequency of breaking changes.

Table VI details the metrics related to each characteristic and the respective *p-values* and *d* coefficients obtained for *top* and *bottom* libraries. Metrics in bold have *p-value* < 0.05, and $d > 0.147$, *i.e.,* they are statistically significant different with at least a small effect size in *top* and *bottom* libraries. As can be observed in the table, the selected *top* and *bottom* libraries are statistically significant different in 6 out of the 12 metrics. The effect size is small in three metrics (number of watchers, number of API elements, and number of releases), and medium in other three (number of watchers, number of contributors, and average API elements per contributor). Next, we analyze each group:

- *Popularity.* Libraries with higher measures for *number of watchers* are on the *bottom* group, *i.e.,* they have higher values of breaking changes. Thus, our results suggests that popular libraries (at least, in number of watchers) are more likely to break compatibility. This contradicts our initial conjecture, once we believed that

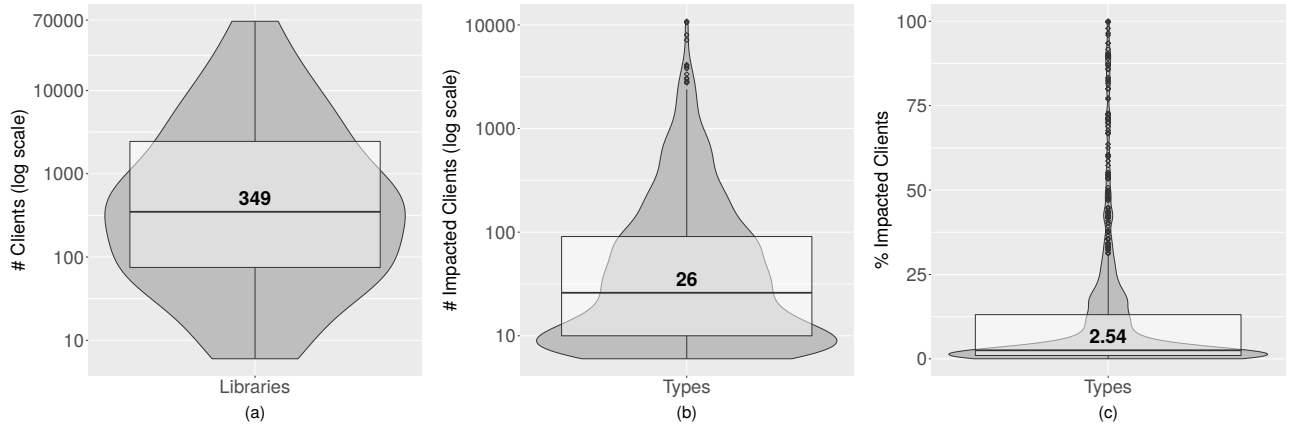

Fig. 5. Impact of API breaking changes in client applications: (a) number of clients of APIs with breaking changes, (b) number of clients impacted by each type with a breaking change, and (c) relative number of clients impacted by each type with a breaking change.

popular libraries would be more careful before inserting breaking changes. In fact, based on these results, we hypothesize that popular libraries have more pressure to evolve, including the need to make design decisions that lead to breaking changes.

- *Size.* Libraries with higher measures for *number of API elements* are also on the *bottom* group. Indeed, libraries with more API elements tend to be harder to maintain and evolve, increasing the probability of breaking changes.
- *Community.* Libraries with higher measures for *number of contributors* and *average API elements per contributors* also appear on the *bottom* group. Thus, our results suggest that libraries with more contributors tend to have more breaking changes.
- *Activity.* Libraries with higher measures for *number of commits* and *number of releases* are on the *bottom* group. Thus, our results suggest that more code changes are more likely to break compatibility.
- *Maturity.* Finally, we detected that there is no statistical significant difference between *top* and *bottom* libraries with respect to their *age* (in number of days).

Summary: Bottom libraries are statistically significantly different from top ones in 6 out of 12 metrics. Libraries with more contributors and more API elements per contributor have more breaking changes, with medium effect size. Also, the number of API elements, the number of commits, and the number of releases affect breaking changes, with small effect. Maturity, though, has no effect on breaking changes.

V. Summary and Findings

1. Libraries often break backward compatibility. We show that 27.99% of all API changes break backward compatibility. On the median, the percentage of breaking changes per library hits 14.78%. In this context, we observe that API breaking changes are recurrent and occur in a relevant percentage. This may occur due to several reasons, for example, (i) unawareness of breaking change risks, (ii) development by naive or less experienced programmers, or (iii) need to restructure the library and, consequently, change the API elements. Therefore, we point out the need for further investigation on reasons developers break contracts with client applications.

2. Breaking changes frequency increases over time. Our study shows that the percentage of breaking changes tends to increase over time by a rate of 20% when comparing their first and fifth years (from 29.02% to 49.14%). This result shows that as time passes, libraries do not become more reliable and stable, as expected. Thus, we suggest the adoption of historical analysis by library developers to measure library stability; this is also important to pressure these developers to avoid compatibility faults. This analysis would also provide useful information for client developers when reasoning whether to depend or not on a library.

3. Most breaking changes do not have a massive impact on clients. Despite the high number of verified breaking changes, we observe that, on the median, only 2.54% of clients are potentially impacted. This low percentage may indicate that (i) library developers pay especial attention on the usage of types before breaking contracts or (ii) the changed types are for internal usage, *i.e.,* not intended to be used by client applications. However, the ratio of impacted clients increases to 13% in a quarter of the studied libraries. Moreover, the analysis of outlier values shows that this impact can be very large, reaching 100% of clients in some cases. Based on that, an impact analysis tool can be helpful for library developers to support their decisions before changing highly used APIs.

4. Development and social coding measures are associated with API breaking changes. We show that libraries with higher frequency of breaking changes have specific project characteristics. We found statistically significant higher values for the following metrics: *number of watchers*, *number of API elements*, *number of contributors*, *average API elements per contributor*, *number of commits* and *number of releases*. Thus, libraries with higher frequency of breaking changes are larger, more popular, and more active. Moreover, notice that

TABLE VI
METRICS AND THEIR RESPECTIVE *p-values* AND *d* ON *top* AND *bottom* LIBRARIES. BOLD MEANS *p-value* < 0.05 (STATISTICALLY SIGNIFICANT DIFFERENT) AND $d > 0.147$ (AT LEAST A SMALL EFFECT SIZE). DIRECTION: "↑" = *top* LIBRARIES HAVE SIGNIFICANTLY HIGHER VALUE ON THIS METRIC. "↓" = *bottom* LIBRARIES HAVE SIGNIFICANTLY HIGHER VALUE ON THIS METRIC.

Dimension	Metric	p-value	d-value	Size	Direction
Popularity	number of stars	0.490	0.272	small	↓
	number of watchers	**0.016**	**0.377**	**medium**	↓
	number of forks	0.679	0.247	small	↓
Size	number of files	0.010	0.017	negligible	↓
	number of API elements	**< 0.001**	**0.149**	**small**	↓
Community	**number of contributors**	**0.014**	**0.330**	**medium**	↓
	average files per contributor	0.454	0.192	small	↓
	average API elements per contributor	**< 0.001**	**0.335**	**medium**	↓
Activity	**number of commits**	**< 0.001**	**0.219**	**small**	↓
	number of releases	**0.001**	**0.251**	**small**	↓
	average days per release	0.003	0.088	negligible	↑
Maturity	age (in number of days)	0.350	0.215	small	↓

the relative measure on the workload of API elements per contributor is also associated with high frequency of breaking changes: the more API elements a contributor has to maintain, the more unstable is likely to be the library. Thus, we suggest the usage of relative development metrics (such as *average API elements per contributor*) as a proxy to developers assess the "health" of their libraries.

VI. THREATS TO VALIDITY

A. Construct Validity

Construct validity is related to whether the measurements in the study reflect real-world situations.

Classification of Repositories. One possible threat of our study is that repositories may have been incorrectly classified into *library* and *non-library*. *Non-library* systems in our studied dataset may bias the results obtained. However, an especial attention was dedicated to this manual classification through the analysis of each repository web page and documentation.

Historical Analysis. In our historical analysis, we consider the first five years of each studied library which represents the third quartile of their age (5.2 years). Therefore, this value can be considered a representative threshold, although not covering the entire life cycle of the studied libraries.

Impact Analysis. To calculate the impact of breaking changes, we count the number of client applications that feature an *import* statement to types that hold a breaking change. A known threat of this decision relates to the impact of breaking changes in fields and methods, since an *import* to their enclosing type does not implies in real usage. However, this measure at least represents the worst case scenario.

B. Internal Validity

Internal validity is related to uncontrolled aspects that may affect the experimental results.

Parser Implementation. A possible threat is the possibility of errors in the implementation of our API *diff* tool, which identifies breaking and non-breaking changes in Java API elements. However, to mitigate this threat, the implementation of *diff* is largely based on a well-known Eclipse library: JDT.

Findings Validation. We paid special attention to the appropriate use of statistical tests (*i.e.,* Mann-Whitney test and Cliff's Delta effect size), specially when reporting the results in *RQ4*. This reduces the possibility that these results are due to chance.

Association and Causation. In *RQ4*, we examined whether there are metrics correlated with high and low frequency of breaking changes. However, it is important to acknowledge that correlation does not imply causation [22].

C. External Validity

External validity is related to the possibility to generalize our results. We focused on 317 popular Java libraries hosted in GitHub, the most used code repository nowadays. Therefore, they are credible and representative case studies, with source code easily accessible. In addition, our client applications dataset has more than 260K Java systems. Despite these observations, our findings—as usual in empirical software engineering—cannot be generalized to other libraries, specifically those implemented in other programming languages. Moreover, we only consider syntactical breaking changes, which result on compilation errors. Behavioral API changes are outside the scope of this paper.

VII. RELATED WORK

API evolution and stability have been largely studied in the literature. Many approaches were proposed to support this activity and reduce its costs for client applications. Dig and Johnson [14], for instance, advance the understanding of API changes, providing basis for designing migration tools. They define a catalog of *breaking changes* and *non-breaking changes*, and, as a result, they found that 80% of the changes that break client systems are refactorings. In this study, we apply this catalog to investigate the frequency of breaking changes between API versions.

Raemaekers *et al.* [3] present four stability metrics based on method changes and removals. The authors investigate their metrics behavior by performing a historical analysis of

stability and impact on 140 clients of the Apache Commons Library. They focus on the clients history, observing the usage frequency and updates in their Maven build files. By contrast, in this work, we study the evolution of a larger set of libraries and compute the impact of breaking changes in an ultra-large dataset of client applications. Additionally, we identify a set of characteristics related to development and social coding that are associated with API breaking changes.

Jezek *et al.* [23] use a dataset of 109 Java programs and 564 program versions to analyze binary compatibility in the context of OSGi-based systems[5]. They define a set of specific changes that may lead to unexpected runtime behavior during "hot upgrades". In this context, the authors conclude that API instability is a common phenomenon, and also that only in a few cases it affects clients. In this work, we focus on source code compatibility, analyzing syntactic changes on API code (which represents the majority of breaking change scenarios).

Thung *et al.* [24] perform a case study to understand how developers reason about and apply changes in three software ecosystems: Eclipse, R/CRAN, and Node.js/npm. As a result, the authors state the differences in practice, polices, and tools applied when performing/avoiding a breaking change. They conclude that in Eclipse developers do not break APIs; in R/CRAN, they reach out affected clients; and in Node.js/npm, they increase the major version number. We highlight that the reasons pointed by the authors on why/how API developers avoid breaking compatibility may explain the low impact of breaking changes found in our experiments.

In the Android context, McDonnell *et al.* [25] investigate API stability and adoption. The authors state that Android APIs evolve faster than client migration. Linares-Vásquez [26] analyze how the number of questions in StackOverflow increases when API are changed. They show that Android developers are more active when they face API modifications.

There are also studies in the context of API deprecation. Robbes *et al.* [6] investigate the impact of deprecation in a Smalltalk ecosystem. They find that some API deprecation have large impact on client applications and that deprecation messages usually have low quality. In the same ecosystem, Hora *et al.* [7] study the impact of API replacement and improvement messages. The results show that a large amount of clients are affected by API changes but most of them do not react. Raemaekers *et al.* [27] investigate deprecation tag usage when studying the frequency of breaking changes on major, minor and patch API versions. The authors observe that methods are commonly deleted without applying deprecation tags, and also that methods with deprecated tags are never deleted. Recently, Brito *et al.* [8] measure the usage of deprecation messages at a large-scale level and propose a tool to recommend these messages.

To support client applications migrating between library versions, some tools and techniques are proposed by the literature. For example, Kim *et al.* [28], present a solution to automatically infer rules from structural changes, computed from modifications on method signatures. Kim *et al.* [29] propose LSDiff, a tool to compute differences between two library versions. Nguyen *et al.* [30] propose LibSync, a tool that uses graph-based techniques to support developers migrating between library versions. Henkel and Diwan [9] present CatchUp, an approach to capture and replay refactoring. Hora *et al.* [13] present apiwave, an approach to keep track of API evolution by mining import statement changes.

Finally, Dagenais and Robillard [31] present an approach that recommends API replacements based on library changes. In contrast, Schafer *et al.* [32] propose to mine API usage change rules based on client changes. In the same context, Wu *et al.* [5] present an approach to produce evolution rules based on call dependency as well as text similarity analyses. Meng *et al.* [11] propose a history-based matching approach to support library evolution.

Novelty: To the best of our knowledge, this work is the largest empirical study investigating API breaking changes and their impact on client applications. Moreover, it also reveals development and social coding characteristics that impact on the frequency of breaking changes.

VIII. Conclusion

This paper presented a large-scale empirical study about API breaking changes. We applied historical and impact analysis to assess: (i) the frequency of breaking changes, (ii) the behavior of these changes over time, (iii) the impact on client applications, and (iv) the characteristics of libraries with high and low frequency of breaking changes. The study was performed in the context of 317 real world Java libraries, 9K releases, and 260K clients. Four research questions were investigated to support library/client developers in maintenance activities. The lessons learned from our experiment results are:

1. Libraries often break backward compatibility. We show that 27.99% of all API changes break backward compatibility. On the median, 14.78% of the changes, per library, are breaking changes.

2. Breaking changes frequency increase over time. Comparing the first and fifth years of the studied libraries, the percentage of breaking changes increase 20% (from 29.02% to 49.14%).

3. Most breaking changes do not have a massive impact on clients. Despite the high number of breaking changes verified, only 2.54% of clients are potentially impacted (on the median). However, this ratio reaches 100% for outlier values.

4. Development and social coding measures are associated with API breaking changes. We found that libraries with high frequency of breaking changes are larger, more popular, and more active.

As future work, we plan to extend this research to other systems, ecosystems, and programming languages. We also plan to analyze other API characteristics (*e.g.,* repository owners, library domain, etc). Another extension of our study is an in-depth qualitative analysis to understand the reasons

[5]http://www.osgi.org

why developers break API contracts. Finally, we intend to implement a tool that may support API developers to assess the impact of API breaking changes.

ACKNOWLEDGMENT

This research is supported by CNPq and FAPEMIG.

REFERENCES

[1] D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software," in *33rd International Computer Software and Applications Conference (COMPSAC)*, 2009, pp. 287–292.

[2] S. Moser and O. Nierstrasz, "The effect of object-oriented frameworks on developer productivity," *Computer*, vol. 29, no. 9, pp. 45–51, 1996.

[3] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *28th International Conference on Software Maintenance (ICSM)*, 2012, pp. 378–387.

[4] M. Reddy, *API Design for C++*. Morgan Kaufmann Publishers, 2011.

[5] W. Wu, Y.-G. Gueheneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 325–334.

[6] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a Smalltalk ecosystem," in *20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[7] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to API evolution? The Pharo ecosystem case," in *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 251–260.

[8] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems," in *23rd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2016, pp. 360–369.

[9] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support API evolution," in *27th International Conference on Software Engineering (ICSE)*, 2005, pp. 274–283.

[10] C. Kingsum and D. Notkin, "Semi-automatic update of applications in response to library changes," in *12th International Conference on Software Maintenance (ICSM)*, 1996, pp. 359–379.

[11] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 353–363.

[12] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain specific warnings: are they any better?" in *28th International Conference on Software Maintenance (ICSM)*, 2012, pp. 441–450.

[13] A. Hora and M. T. Valente, "apiwave: keeping track of API popularity and migration," in *31st International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 321–323.

[14] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," in *22nd International Conference on Software Maintenance (ICSM)*, 2006, pp. 83–107.

[15] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 422–431.

[16] ——, "Boa: ultra-large-scale software repository and source-code mining," *Transactions on Software Engineering and Methodology*, vol. 25, no. 1, pp. 1–34, 2015.

[17] R. Grissom and J. Kim, "Effect sizes for research: a broad practical approach," *Lawrence Erlbaum Associates Publishers*, 2005.

[18] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? A case study on free Android applications," in *31st International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 301–310.

[19] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "API change and fault proneness: a threat to the success of Android apps," in *9th International Symposium on the Foundations of Software Engineering (FSE)*, 2013, pp. 477–487.

[20] A. Hora, M. T. Valente, R. Robbes, and N. Anquetil, "When should internal interfaces be promoted to public?" in *24th International Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 1–12.

[21] J. Businge, A. Serebrenik, and M. G. van den Brand, "Eclipse API usage: the good and the bad," *Software Quality Journal*, vol. 23, no. 1, pp. 107–141, 2013.

[22] C. Couto, P. Pires, M. T. Valente, R. Bigonha, and N. Anquetil, "Predicting software defects with causality tests," *Journal of Systems and Software*, vol. 93, pp. 24–41, 2014.

[23] K. Jezek, J. Dietrich, and P. Brada, "How Java APIs break–an empirical study," *Information and Software Technology*, vol. 65, no. C, pp. 129–146, 2015.

[24] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: cost negotiation and community values in three software ecosystems," in *24th Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 1–12.

[25] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *29th International Conference on Software Maintenance (ICSM)*, 2013, pp. 70–79.

[26] M. Linares-Vásquez, G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "How do API changes trigger stack overflow discussions? A study on the Android SDK," in *22nd International Conference on Program Comprehension (ICPC)*, 2014, pp. 83–94.

[27] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the Maven repository," in *14th Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2014, pp. 1–10.

[28] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *29th International Conference on Software Engineering (ICSE)*, 2007, pp. 333–343.

[29] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 309–319.

[30] H. A. Nguyen, T. T. Nguyen, J. Gary Wilson, A. T. Nguyen, M. Kim, and T. N.Nguyen, "A graph-based approach to API usage adaptation," in *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010, pp. 302–321.

[31] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 481–490.

[32] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 471–480.