An Exploratory Study into how Web Service API Evolution Impacts API Clients

Tiago Espinha, Andy Zaidman, Hans-Gerhard Gross

Report TUD-SERG-2013-011
An Exploratory Study into how Web Service API Evolution Impacts API Clients

Tiago Espinha, Andy Zaidman, Hans-Gerhard Gross
Delft University of Technology, The Netherlands
{t.a.espinha, a.e.zaidman, h.g.gross}@tudelft.nl

Abstract—Web services provide a systematic and extensible approach for application-to-application interaction. Developers using web service APIs are forced to accompany the API providers in their software evolution tasks. In this paper, we investigate how major web service providers organize their API evolution, and we explore how this affects source code changes of their clients. Our exploratory study of the Twitter, Google Maps and Facebook APIs analyzes the state of API evolution practices and provides insights into the impact of service evolution on client software. Our study is complemented with a set of observations regarding best practices for web service API evolution.

I. INTRODUCTION

Modern-day software development is inseparable from the use of Application Programming Interfaces (APIs) [1]. Software developers access APIs as interfaces for code libraries, frameworks or sources of data, to free themselves from low-value programming tasks and/or speed up development [2]. Web service APIs offer a systematic and extensible approach to integrate services into (existing) applications [3].

However, what happens when these APIs start to evolve? Lehman and Belady emphasize the importance of evolution for software to stay successful [4], and updating software to the latest version of its components, accessed through APIs [5]. Dig and Johnson state that breaking changes to interfaces can be numerous [5], and Laitinen says that, unless there is a high return-on-investment, developers will not migrate to a newer version [6]. This leads to a dichotomy between library developers who want to release new versions of their software with new features and bug fixes, and client developers, who may be reluctant to adapt their applications to the changes made in the APIs [7]. Ideally, an API should be built such that it is never broken, but Raemaekers et al. indicated that this may prove impossible in practice [1]. Similarly, Dagenais and Robillard witnessed that breaking changes can range from simple refactorings to a complete rearchitecture of the API [2].

In the case of statically linked APIs, e.g., an XML library that is used in a Java program, the software development team has the choice to immediately upgrade to the newest version or to postpone migration. The trade-off is the extent to which the latest functionality indeed brings return-on-investment.

Web services present a different picture. Even though API provider and client are much more loosely coupled in terms of service execution, in terms of API evolution they are tightly dependent. When the service provider changes the API, this upgrade is forced upon all the API clients. Unless special actions are taken by the provider to ensure backwards compatibility, the clients may break upon facing new functionality or they may simply stop working altogether, e.g. a new network address is made available for the new version and the old one removed. This concern was already studied by Raemaekers et al. who established that backward compatibility is a major concern for any library developer [1]. A fundamental problem in using web services is, therefore, the fact that client developers can no longer choose when to upgrade to a newer version of the API provided.

In this paper, we investigate how web service APIs evolve, and study the consequences for clients of these web services. We look at the following research questions:

RQ1 What are some of the policies that high-profile public web service providers follow for upgrading to newer versions of their APIs?
RQ2 What is the impact of web service API changes in terms of code churn in the client applications?
RQ3 What is the nature of the source code changes in clients caused by web service API evolution?

The exploratory study presented investigates three high-profile web service APIs, namely those of Facebook, Twitter and Google Maps. For Twitter and Google Maps we investigate 3 projects from each, whereas from Facebook we investigate 1 project.

The remainder of this paper is structured as follows: in Section II we describe the different API policies from the different providers, Section III describes our experimental setup including how the projects were selected, what is an API change and how to find it, Section IV presents our results and Section V frames these results with our research questions and provides a list of recommendations for API providers. Lastly, we discuss related work in Section VI and present our conclusions in Section VII.

II. EXPLORING API POLICIES

Our study relies on projects using three web service API providers: Facebook, Twitter and Google Maps. We specifically targeted these providers as the APIs are used in many open source projects, i.e. we found 691 projects using Facebook, 719 projects using Google Maps, and 252 projects using Twitter. Additionally, all three of them have had to evolve at least once. How the client projects were selected is further discussed in section III-A.

Our first research question is concerned with the respective policies applied in API evolution, because this determines how
changes will be “pushed” to the clients. Here, an interesting issue is how deprecation is managed, and how much time is given to developers to upgrade. This second aspect is particularly relevant since, despite all the services are provided through REST over HTTP, the way changes are planned and deployed is very different across providers. In the following, we look at each of them individually.

A. Google Maps

The Google Maps’ API falls under the global Google deprecation policy, i.e., whenever products are discontinued or backwards-incompatible changes are to be made, Google will announce this at least one year in advance. Exceptions to this rule regard whenever it is required by law to make such changes or whenever there is a security risk or “substantial economic or material technical burden”. To summarize, save for security-related bugs, Google will provide a 1-year window for the transition to a new API.

In practice, however, Google is much more lenient, e.g., analyzing the migration Google Maps version 2 to version 3, Google provided a 3-year period for this transition rather than the standard 1-year deadline. Additionally, before the deadline Google prolonged this period for another 6 months, effectively offering a 3.5-year period for the transition. Why they offered such long period is not certain. However, anecdotal evidence from Google’s user forums shows that many developers waited until the last moment to upgrade. In March 2013, an unnamed developer asked “I’m working on upgrading to v3 but I’m expecting to finish 2 or 3 weeks after 19 May [initial deprecation date], so I was wondering if we can get an official answer about this”. Similarly, when earlier in 2013 Google experienced an outage in all its Maps APIs’ versions, several developers also asked whether v2 had already been taken offline, thus revealing that a number of developers were still using it. Google’s provision of a very long transition period may have led the developers to be too relaxed about the deprecation, leading them to migrate at the latest moment.

B. Twitter

Twitter has no official deprecation policy. However, the announcement for the current API version set a 6-month period to adjust to the change. Since it implies a different endpoint URI, both versions could in fact be maintained in parallel indefinitely, and it means that once the old endpoint is disabled, all applications using it will break. Despite the 6-month period Twitter did not follow the original plan. The new API version, announced in September 2012, was intended to fully replace the old version by March 2013. However, rather than fully take it offline, they decided to approach the problem by starting to perform “blackout tests”, both on the date the API was supposed to be taken offline and twice again two weeks apart after the original deadline. These blackout tests last for a period of one hour and can occur at random during the days they are announced, and they act as indicator for reluctant users, that they should migrate.

This approach contrasts with that of Google and Facebook but gathers appreciation in its own right. The blackout tests have been very well received, with developers claiming “These blackout tests will be super helpful in the transition. Thanks for setting those up!”

C. Facebook

Facebook’s approach to API evolution is substantially different as they do not use an explicit versioning system. Instead, the introduction of new feature is done by an approach referred to as "migrations,” which consists of small changes to the API that each developer can enable/disable at will during the roll-in period. After this period they become permanent. The Facebook Developers website claims that Facebook provides a 90-day window for breaking changes. Like Google’s, this policy also explicitly excludes security and privacy changes, which can come into effect at any time without notice.

Facebook is also in the process of changing this policy. While, so far, there have been breaking changes put into place every month from January 2012 to May 2013 (with the exception of March 2012), Facebook has announced that from April 2013, all the breaking changes will be bundled into quarterly update bulks (except security and privacy fixes).

In addition, Facebook has an automated alert system in place, which e-mails developers whenever the features they use are affected by a change. Dynamically determining which developers are relying on which features of the API goes along our previous line of work [8] where we investigated to which extent such a mapping affects system maintenance.

III. EXPERIMENTAL SETUP

A. Project selection

We chose three major API providers to ensure access to a large code base of open source projects using their APIs. This implied that for each provider several hundred projects exist which are potential candidates for our analysis. However, because our analysis requires projects accessing at least two versions of an API we automated a filtering mechanism to find projects which meet these conditions. We first compiled a list of all the projects available on GitHub that use the latest versions of these APIs, and then selected the projects which also contained references to the old API. This approach worked well for Twitter and Google Maps where a clear cut version upgrade was imposed by the providers. However, because Facebook relies on small incremental upgrades rather than versioning of the API, another approach was required.

---

1Deprecation is a status applied to a computer software feature, characteristic, or practice indicating it should be avoided, typically because of being superseded... from http://en.wikipedia.org/wiki/Deprecation
2Google Maps Terms — https://developers.google.com/maps/terms
3Twitter API v1.1 — https://dev.twitter.com/docs/api/1.1/overview
4https://dev.twitter.com/blog/planning-for-api-v1-retirement
5Facebook’s approach to API evolution is substantially different as they do not use an explicit versioning system. Instead, the introduction of new feature is done by an approach referred to as "migrations,” which consists of small changes to the API that each developer can enable/disable at will during the roll-in period. After this period they become permanent. The Facebook Developers website claims that Facebook provides a 90-day window for breaking changes. Like Google’s, this policy also explicitly excludes security and privacy changes, which can come into effect at any time without notice.
6Facebook is also in the process of changing this policy. While, so far, there have been breaking changes put into place every month from January 2012 to May 2013 (with the exception of March 2012), Facebook has announced that from April 2013, all the breaking changes will be bundled into quarterly update bulks (except security and privacy fixes).
7In addition, Facebook has an automated alert system in place, which e-mails developers whenever the features they use are affected by a change. Dynamically determining which developers are relying on which features of the API goes along our previous line of work [8] where we investigated to which extent such a mapping affects system maintenance.
1) Twitter: For the Twitter case-study, we looked for a string containing the latest API endpoint as follows “https://api.twitter.com/1.1/”, as this is the only way for a client application to use the latest version of the Twitter API.

2) Google Maps: In the Google Maps case-study, the approach was similar to that used for Twitter. The changes to the API were extensive to the point where the simple instantiation of a map required a new constructor. We searched for the constructor used in version 3 of the Google Maps API, that is new google.maps.Map.

3) Facebook: Facebook does not have a strict versioning mechanism. Instead, changes are pushed in the form of migrations which touch small pieces of functionality. Selecting candidate projects was done by searching for the API endpoint (https://graph.facebook.com/) which has remained unchanged throughout the evolution of the API.

B. What is an API change and how to find it?

We used the same general approach (with slight variations) across all API clients to detect API changes: finding projects that make use of the new API that also contain(ed) references to the old API. In the case of Twitter this change corresponds to finding references to the old endpoint (https://api.twitter.com/1/), or in the case of Google Maps the commit where the old constructor for version 2 (new GMap2) was used. Facebook required a slightly different approach. Because the evolution is not a clear cut migration, we searched for the small pieces of functionality that throughout time were deprecated or changed by Facebook (as reported by Facebook in the completed changes feed). In practice, we first used a script that filters projects for their use of the latest API. A second script searches for references to the old API. In all the diffs of the projects found in the first step.

C. Impact evaluation

The goal of the paper is to investigate how web service APIs evolve, and how this affects their clients. So, for each project we looked at the commits right before and right after the first commit containing references to the new version of a provider. This was done to identify potential initial preparations prior to bringing a new API online, as well as to check for potential fallout effect caused by switching to the new API.

In order to estimate the amount of work involved in maintaining the clients of a web service API, we start by using the code churn metric [9], which we define for each file as

\[
\text{FileCodeChurn} = \frac{\text{LOCAdded} + \text{LOCChanged}}{\text{TotalLOC}}
\]

The code churn we analyze and display in the graphs below represents the code churn for each commit. It allows us to discriminate the code changes in the clients required to address an API upgrade with code change for other software maintenance tasks. We identify which commit first introduced the reference to the new API version, and then investigate the subsequent commits for API-related maintenance. Knowing how much code churn was involved in the API evolution, and comparing this value to the average for the project, allows us to assess the amount of work related to API maintenance. By looking at the graphs (presented in the following section), we also visually investigate whether there are abnormal variations in the churn values in the time period preceding and succeeding the API change. When that is the case, we analyze the outlier commits and attempt to identify API-related changes.

While code churn provides a good starting point for assessing the impact of a maintenance task, it does not provide the whole picture: the nature of the code change, the number of files involved and their dispersion also help in determining the impact of a change. Hence, we also provide a more in-depth view of how the API migration affects a particular project. This is done by looking at the number of source code files changed, and analyzing the nature of the changes (e.g. file dispersion, actual code changes, whether the API-related files are changed again). This analysis also allows us to mitigate the code churn’s indifference to the complexity of code changes.

IV. Results

In this section we present the results for the projects under analysis, divided under their respective API providers. For each project, we present a graph with the average code churn per commit and explain our findings. We start by identifying the commit(s) where the migration to the new API has taken place and investigate how the code churn compares with other commits. Secondly, we look at how the changes have actually impacted the source code: the number of files touched, the dispersion of changes across the code. Lastly, we investigate the code churn peaks as to why these happened and whether they are related to the API evolution task.

A. Twitter

Three of the projects under analysis make use of the Twitter API and underwent maintenance to migrate from from API version 1 to API version 1.1. The change is done by changing the REST API endpoint URL, which means that for a period of time, all the projects which did not change the endpoint URL would keep using the old version of the service.

However, the new version also installed deeper changes which make the evolution task more complex than just switching the endpoint URL. Examples of such changes are a) all requests to the Twitter API requiring authentication and b) the dropping of the XML format altogether. A less invasive change is related to the new rate limiting rules imposed.

Despite all the changes, the change to the Twitter API has been done in one clean swift move. The announced changes were made available in a new API endpoint and during a six month period, developers could adjust their clients to use the new API. While the final deprecation date of the old API has not yet been reached at the time of writing (June 11, 2013), as described before in Section II-B Twitter put in place “blackout tests” three times before the deprecation date. These tests, meant as an eye opener for the developers, were well received with the developers claiming “These blackout tests will be super helpful in the transition. Thanks for setting those up!”.

Our analysis focuses on three projects that have already migrated to the new API version. The code churn of these projects can be seen in Figures 1, 2 and 3. In all our figures the API-related commits are highlighted in a different color and even though the x-axis represents a timestamp, it should be noted that the spacing does not reflect temporal distance to scale; each bar represents one commit. Also to facilitate the visual comparison of the code churn landscape, the code churn scale was trimmed to a maximum of 0.02 in every project.

The larger peaks were a cause of intrigue since those particular commits represent a much larger code churn than any of the other commits. It was also not known whether these peaks could be major changes paving the way for the final change which puts the new API client in place, and for this reason we manually analyzed these commits to determine what exactly is being changed. Similarly, we analyzed the commits in the temporal vicinity of the commit we identified as enabling the new API.

In this section we repeatedly refer to hexadecimal hashes. Because all projects were obtained from GitHub, each of these hashes uniquely represents a commit in that specific project.

**rss twi2url** — The rss twi2url project is an application written in Javascript which continuously parses a Twitter feed into an RSS feed. It consists of four main source files and five configuration and library files.

a) **Code Churn**: In this particular project (Fig. 1), the migration to the new API is done in one single commit which is also the very last commit to date (e98815e). The code churn for this particular commit has a value of 0.008251 which places it slightly above the average without file addition (0.007250493) and slightly below the average with file addition (0.008393189). This project’s API evolution commit hovers around the average code churn for the remaining commits. Considering the lower file and line count of this project, we consider this code churn low as the changes caused by the API did not cause major changes to the code base.

In general, this project presents a higher code churn than projects with a larger absolute line count. This is due to the fact that each change in relation to the total number of lines represents a higher code churn percentage. This should be taken into consideration when analyzing the code churn graph.

b) **Impact analysis**: The source code changes are generally small and are covered in three files, one of which simply adjusts the fetch frequency from Twitter’s API. While it is easy to assume that the low code churn is attributed to good architectural practices, such phenomenon could also be attributed to a small code base.

c) **Code Churn Peaks**: We also investigated the high code churn peaks in an attempt to understand whether they could be somehow linked to the API evolution task. The commit with the highest code churn (c3a2b09, code churn of 0.48696) represents a heavy refactoring commit which changed nearly half of the project’s code base. Even through manual inspection it is still unclear to whether these changes are related to the API evolution; the changes are extensive and reach out to nearly all the project’s code which makes it difficult to understand what is the motivation for the refactoring.

The second highest (4fb04d8e, code churn of 0.278188) is a simple file rename which counts towards code churn in our approach but is irrelevant to the API discussion. The third highest commit (2fd12fe, code churn 0.143787) consists of changes to how the application deals with parallel connections and has no connection to the new API.

**TwiProwl** — The TwiProwl project is written in Ruby and it consists of a script to generate push notifications on iOS out of a Twitter feed. It is relatively small (3 source files, 966 LOC) since it is a mere script.

a) **Code Churn**: The code churn for this project is generally high (Fig. 2) which is explained by its relatively small code base. Any small change to the script represents a high code churn in percentage. Despite its small size, it has been under continuous development since 2009 and contains contributions from more than one developer.

As far as evolution goes, the TwiProwl project also follows a single-commit evolution behavior. In fact, the commit which brings the API changes (83034c9e, code churn 0.030629) has a time-gap of more than two years from the previous commit. The fact that the previous commit happened before Twitter changed its API rules out any previous high code churn peaks as potential refactorings to accommodate the API change. We also analyzed the posterior commits which consisted entirely of changes to the README file and to configuration files, all of which unrelated to the API change.

The commit which contains all the API changes also presents a higher code churn than average (average 0.02191286 versus 0.030629). Upon manual inspection, the changes do seem to all be linked with the API evolution task which is further supported by the commit message which is titled “update API endpoint to 1.1” and contains a description of the changes made.

---

10 rss twi2url — https://github.com/take-cheese/rss_twi2url

11 TwiProwl — https://github.com/takuo/TwiProwl
b) Impact analysis: The API evolution task in this project is done on one single file. Then again, the project consists of only three source files. Because the bulk of source code is concentrated in one single file (87% of LOC), this is to be expected. Due to the small size of the project and the nature of the application (Twitter notification script), not much can be said regarding the impact of the API maintenance task.

c) Code Churn Peaks: While this project has a high average code churn, we still investigated the three highest code churn peaks in an attempt to identify which tasks take up most of the churn for this particular project.

The highest code churn peak (8ced0a5) comes at 0.705264 which means that roughly 70% of the code changed. Some of the changes are cosmetic, some address uncaught exceptions and some simply change the flow of the script. Most code churn, however, comes from the addition of new functionality.

The second highest peak (e1050df) with a code churn of 0.509 only contains file additions. Manual inspection showed that the added files are in fact a direct copy of a library to deal with command line input.

Lastly, the third peak (ce776cb) with a code churn of 0.340426, turned out to be the addition of a logging mechanism as opposed to logging to stdout.

From our peak analysis we conclude that none of these commits are related to the API evolution task.

**netputweets** — The netputweets project is a Twitter web client for mobile phones. It is written in PHP. It presents a slightly different evolution picture compared to the other Twitter-based projects. Namely, the API evolution was dealt with in a wave of three commits rather than all at once.

a) Code Churn: Regarding code churn (Fig. 3), the evolution was kicked off by commit 882943d (code churn 0.001241) which is the first commit in a period of three months (thus ruling out any previous preparation). Half a day later, commit e36c9f0 (code churn 0.000790) brings several changes related to the API change. More specifically, this commit fixes the way the client deals with the data provided by the API which seems to have been broken with the initial commit.

The last commit (5510a0d) changes 40 references in 4 files to a static variable changed in the previous commits. The changes are small and consist of a small variable substitution.

Important to note is how most of the API changes regarding netputweets are effected using change commits rather than adding files. In fact, only one file is added. The file is added in 882943d which is a configuration file to cope with the new authentication policies of Twitter.

b) Impact analysis: While commit 882943d involves 10 files (9 changed and 1 added), only 5 of the modified files are actually related to the API evolution. The changes to these files are also extensive and contribute the most to the measured code churn. Changes in these 5 files are made to the endpoint URL, to the way Twitter posts are paginated, to the way direct messages, search and timeline are accessed and a transition is done to the JSON format as the XML format is deprecated. All these changes seem to be a direct result from changes in the behavior of Twitter’s API.

The next commit, e36c9f0, contains API related changes in one of the three files involved. Once again, the major code churn contributor is the one related to the API. Here more changes are made to how users are searched and how the results from the search are paginated, as well as how the users’ information is displayed. While only one file is touched, the most likely case was that this file was forgotten in the first round of API maintenance. Possibly only remembered when unexpected behavior was found by the developers who had already switched to the new endpoint.

c) Code Churn Peaks: Analyzing the high code churn peaks in the netputweets project we concluded they were not related to the API maintenance task. In fact, the first API-related commit happened four months after the previous commit, which leads us to believe there was no preparation that led to the maintenance job. The highest code churn peak at 0db5c25 involves 3 source files. Manual inspection revealed no connection whatsoever to the API evolution.

B. Google Maps

In order to analyze the projects which use the Google Maps API, we chose the most recent breaking change in this API's
evolution: the transition from version 2 to version 3. This particular evolution requires an entirely new approach in order to do something as basic as instantiating a map. For this reason, we expected all the projects which rely on this API and which are maintained to have undergone this evolution.

**hobobiker** — The hobobiker\(^{13}\) project consists of a website written in PHP created to track the traveling of two bikers. It is based on Drupal (a content management system) and has been adapted to fit Google Maps.

a) **Code Churn:** At a glance, the hobobiker project (Fig. 4) contains commits with lower code churn compared to all the other projects analyzed in our paper. The lower code churn is a direct result of the large LOC overhead added by the Drupal code base. With a total LOC count of ~440k lines, the code churn for each commit (calculated based on the total LOC) becomes comparatively lower. Despite the generally low code churn observed, comparing the code churn from the API commits to the project average still reveals how the API changes compare in terms of work to the remaining maintenance tasks.

The API changes are done in a two commit process (699e342 and 854b6b2) with code churn 0.000289 and 0.000161 respectively. Considering the code churn average without file addition (as there were no new files added for the API evolution), we found that the API changes amount to approximately 2.4 times the average churn (0.00018336).

b) **Impact analysis:** In the first of two commits (699e342, code churn 0.000289) 170 lines are changed. As we manually analyzed, the changes consist of cosmetic non-functional changes such as indentation, whitespace and capitalization.

Despite the cosmetic nature of this commit’s changes, they are very likely connected to the API migration. The changes involve the very same files which are changed in a posterior commit with the actual API changes and may have been a way for the developer to better understand the code at hand.

The second commit (854b6b2, code churn 0.000161) also contains API changes diluted in other unrelated changes. This particular commit deals with several changes aimed at improving the existing code (e.g. protection against SQL injection, addition of a centralized logging facility), which caused a significant amount of code to be removed and modified. Out of the 63 changes, only 25 changes are related to the API evolution task as these are the changes that bring in the new version’s Javascript files and which start making use of v3’s constructor. By analyzing the code, the changes are similar to those of the first commit but are performed in a different file which was also making use of the old version of the API. The API-related changes in this commit are then quite small which is justified by this particular project containing only very simple Google Maps’ functionality.

c) **Code Churn Peaks:** The three highest code churn peaks in this project come exclusively from the addition of files. Given the quantity of files added, this points towards the addition of third-party libraries.

The highest code churn peak (5fb9997) comes at 0.004454 which represents a commit that is approximately 40 times larger than the API evolution commit (854b6b2). This commit simply adds an audio library.

The second code churn peak (b25ca60) has a code churn of 0.003273. Once again this commit only concerns the addition of files and it regards the addition of a theme for the content management system platform.

Lastly, the third code churn peak (136d8a7) with a code churn of 0.003144 has a very similar amount of change as the second peak. Manual inspection revealed that the change was once again related to the audio library added in the first peak.

Of note is also the fact that all commits happened in 2009 (approximately 4 years before the API commits). In sum, none of the peaks are actually related to the API evolution.

**cartographer** — The cartographer\(^{14}\) project is a library written in Ruby to facilitate the integration of Google Maps in Ruby on Rails applications. While the library has not been committed to since March 2011, it does in fact already support version 3 of the Google Maps API. Simultaneously, it provides backwards support for version 2. The reason behind this is that this particular project is a software library in its own right, meaning that other projects depend on it. Therefore, the developers of this library added the new version but ensured that backwards compatibility was maintained. In fact, the library still defaults to version 2 of Google Maps as seen through this comment in the lib/init.rb file: “By default we make cartographer to use Google maps v2 for backward compatibility”. Whenever developers want to use version 3 of Google Maps through this library, they must explicitly do so.

a) **Code Churn:** In terms of code churn, the average is fairly low compared to other projects (0.013512057 including file additions). For the API evolution process, the developers pushed all the changes over the course of two commits (3fb14ac and 495633b by this order). We identified commit 3fb14ac as the initial point of the API evolution which is supported by the commit message “Initial Commit for re-write

\(^{13}\)hobobiker — https://github.com/rfay/hobobiker.com

\(^{14}\)Cartographer — https://github.com/joshuamiller/cartographer

![Fig. 4. Code churn per commit hobobiker](image-url)
of lib to support Google Maps v3”. This commit presents a very interesting picture as it reorganizes the existing source code files to allow both maintaining the old version and adding the new one. Because both versions are supported, what were unique source files for dealing with Google Maps are now moved into a “v2” folder. Similarly, a changed copy of the v2 files are brought into the “v3” folder.

This also causes a great discrepancy in the code churn. Including the addition of files, the code churn is almost twice as high as it is by considering only modifications (0.09282 versus 0.054313). In sum, the majority of the changes comes from adding support for v3 by copying files and making small changes to adjust the files to the new version. Some modifications are also made to the existing interface as to ensure backwards compatibility. Code churn for this task (with file additions) is nearly ten times higher than average.

The second commit which brought API-related changes (4db8e97) adds a further 0.022832 code churn. This commit is titled “First functional re-write of Cartographer to support Google Map api V3” and manual inspection confirmed that all the changes are related to the API change. Moreover, the changes seem to be essential for the new API to work. Namely, looking at the previous API-related commit, several files were copied and left unchanged from version 2. In this commit they are then adjusted to accommodate the API changes for v3.

b) Impact analysis: In order to analyze the impact of the API changes for this particular project, we counted all the files that were added for version 3 and the files from version 2 which had to be modified to accommodate the fact that both versions are maintained. Considering this, 10 files were added/changed in total which translates to 9 added files for version 3 and 1 modified file in version 2.

Considering the majority of the churn is caused by adding files and only a small percentage is contributed by modifying existing files, it appears as if the architecture of the library is very stable. Contributing to this is the fact that separate files sets deal with different versions, which helps the separation of concerns for separate versions.

c) Code Churn Peaks: This project contains three major code churn peaks. The highest peak (4db8e9, code churn 0.096794) happened right after the first API-related commit and we did not consider it to be related to the API evolution, it does in fact implement a new feature brought forward to the new version of the API. However, this particular feature is an extra feature which did not exist in the first place.

The second peak is in fact the first API-related commit (3fb14ac, code churn 0.09282) which is already analyzed in the code churn section of this project.

The third peak (8efdac8, code churn 0.042478) happened on the day before the API-related commit and no other commits exist between the two. This commit incorporates code from another library and adds the possibility to manage markers in the Google Map. Again, because this is an entirely new feature and adds significant chunks of code, it justifies the higher than normal code churn.

wohnungssucherportal — The wohnungssucherportal15 is written mostly in Ruby (on top of Ruby on Rails) and as described by the developer, it is an “application to manage the search for a new flat”.

a) Code Churn: This project is characterized by particularly low code churn commits on average. Despite the few peaks in some of the commits, on average, the code churn is low at just 0.00169416 (including file additions). The single commit which implements the transition to Google Maps API version 3 (commit 06169f4) comes at just 0.000127.

b) Impact analysis: The API-related changes have been introduced in a single commit and four files were changed in this commit, all of them related to the API change. In this particular case, even though only four files were changed, the changes concern hardcoded references in separate parts of the application. This may indicate poor architectural design.

We also found two commits in the temporal vicinity of this commit which present a relatively higher code churn (commits fc59bc6 and 8339065, code churn 0.001223 and 0.00087). Because of the higher code churn we decided to investigate what changes had been introduced.

Manual inspection revealed that the first commit (fc59bc6) is actually the moment when the old version of Google Maps API is added (1 file added). This however, represents a very small percentage of the code churn. The high code churn is justified from the addition of an extensive file containing test data (very verbose HTML and Javascript code) for the page scraper unit test. This test data contributes approximately 97% of this commit and was added simply to complement the unit test which is not related to the API evolution.

The second commit (8339065) has no relation with the API change. It consists only of localization files which are verbose and therefore generate an above-average code churn peak.

c) Code Churn Peaks: In terms of absolute peaks, three commits stand above all the rest. The highest peak (b6d6f2b) with a code churn of 0.132653 owes its high churn to the change. It consists only of localization files which are verbose and adds significant chunks of code, it justifies the higher than normal code churn.

![Fig. 5. Code churn per commit cartographer](image-url)
The second highest peak (bcec1dc) with a code churn of 0.020292 touches two files. One of the files is a copy of an existing functional test with significant changes. The large code churn comes from all the changes done to the verbose HTML code done to adjust the test to a different scenario. The changes done to the second file changes the model of the application in order to accommodate the new test. Lastly, the third peak (679e662) with a code churn of 0.01351 introduces a whole new test case (again, in HTML and Javascript which makes it very verbose) and makes changes to an existing test. The changes also include a new model for a “page scraper” and changes to the page scraper itself.

In conclusion, and by looking at all the changes introduced in the major code churn peaks, none of the changes presented contained any ties to the API evolution task.

C. Facebook

Because Facebook API changes are small and more frequent, we needed a different approach for our investigation. Rather than just finding one single API evolution point, we collected a group of breaking changes reported by Facebook and attempted to find these in the commit diffs of all the projects. Even with the adjusted approach, we only found one project which contains one of these breaking changes. While it is unclear why no other projects have undergone API-related evolution not even once, this may be due to the all-encompassing nature of the Facebook API and how the evolution is done in small changes to very specific features. Examples of such small changes are the removal of game-specific attributes and adding required permissions to read mailboxes (which only affects clients reading mailboxes).

spring-social-framework — The spring-social-framework brings the Facebook integration to the J2EE Spring Framework. It is written in Java and contains approximately 29k LOC. Of note about this project is that it has been under development since 2010 and it has constantly been under development (i.e. frequent commits).

a) Code Churn: As far as code churn goes, the API evolution changes came in one single commit which mends functionality broken by a “migration”. The commit (72ce8f88) has a relatively low code churn (0.000276 versus an average of 0.00118451) and it deals with a number of features which were removed and changed in Facebook’s July 2013 migration.

b) Impact Analysis: The particular migration which caused this evolution introduces more than 10 different changes. Only a few of these have affected this project but it is still not surprising that in total 14 files were involved in this change. The removal of version numbers on Facebook groups was contained to 3 files, whereas the change in how Facebook stores photos caused 2 files to be changed. Changes to how geolocation data also caused another file to be changed. The remaining files changed in this commit consist only of unit tests which had to be adapted to reflect the new behavior.

Despite the seemingly high number of changed files, it must be stressed that 10 changes were introduced. Each change that has actually affected this project was in fact handled by touching essential files for that feature which was in all cases architecturally well-abstracted.

The changes also deal with unit tests which reveals concern from the developers in maintaining good code coverage.

c) Code Churn peaks: In this project we identified the three major code churn peaks. The highest code churn peak (fac93f6c) comes in at 0.259663 which represents a change of approximately 26% of the code base. This commit, however, touches only non-functional files. Namely, it touches Maven deployment XML files as well as the CSS files for the Javadoc generation. Also under this commit we found licensing files which consist only of text.

The second code churn peak (fe98367) sits at almost half of the first peak (0.146412). This particular peak does have a small relation to the Facebook API. However, it simply adds a “connect to Facebook” button to a showcase sample within the framework and this only accounts for 0.7% of the code churn. The remaining 99.3% is the addition of the jQuery library.

The third and last code churn peak under analysis (92886ef) comes in at approximately half of the code churn of the previous one (0.060846). The changes are non-functional and address only an XML file which is part of the documentation.
V. Discussion

In this section we use our findings to address our three research questions and present a list of five do’s and don’ts for developers of API web services.

A. Answering the Research Questions

We start by answering the research questions laid out in the introduction regarding the three different API providers.

1) RQ1: “What are some of the policies that high-profile public web service providers follow for upgrading to newer versions of their API?” Our findings show that the policies vary greatly amongst different providers. We chose Facebook, Twitter and Google as their APIs are widely used yet despite the shared popularity, no consensus exists in how API changes should be carried out.

Google provides extensive periods of time (officially 1 year, although 3 years were observed) for developers to migrate to newer versions of their API, whereas Twitter provides a 6 months period for this change. Twitter is, nonetheless, special in the “blackout tests” it provides by temporarily turning off the old API versions to alert developers of the upcoming change. Despite the initial strict deadline for the migration, both providers extended the deadline considerably.

The Facebook API evolution is different as it is done through small changes which are pushed monthly to the developers and which the developers can, for a period of 6 months, enable and disable at will for their particular client. More detail on this question can be found in Section II.

2) RQ2: “What is the impact of web service API changes in terms of code churn in the client applications?”. For each project, we compared the code churn for the API migration to the average code churn for that project. In general, the code churn involved in API maintenance is similar that of day-to-day software maintenance. Our results show that while the dispersion of the changes across the source code varies from project to project (due to different architectural approaches), the impact caused by the web service API changes seems to be no different from that of normal software maintenance.

While the impact of API evolution on code seems to be reasonable, we should remember that the changes are still forced upon the developers of the clients. As such, while it is an unavoidable maintenance task, it also brings the latest security updates and features.

3) RQ3: “What is the nature of the source code changes in clients caused by web service API evolution?”. We observed that the source code changes used to accommodate the web service API evolution vary across projects and depend greatly on the architecture. In our findings, some projects must change every file where the API is used whereas other projects abstract the changes well enough in a subsystem for API interactions.

The nature of the changes seem no different than those of a statically linked API library. However, the main consideration to have when using a web service API is that the changes are inevitable at some point down the road. This indicates that more thought needs to be put into the architecture right from the start to accommodate future API evolution.

B. Recommendations

Based on our investigation and additional insights obtained from observing developer fora. We compiled a list of five recommendations for web service API providers with regard to easing the evolution task for developers of API clients.

1) Do not change too often: Facebook is pushing monthly “breaking changes”, yet a recent survey on API integration pain revealed that this policy has caused distress amongst developers. It is unclear whether this has played a role in Facebook moving to quarterly updates (April 2013).

2) Old versions of the API should not linger too long: Google started off with a 1-year timeframe for the deprecation of Google Maps’s version 2, and ended up extending it to 3 years. Yet, reaching the 3-year mark, many developers still flocked to the developer forums in hopes that the deadline would be extended further (which happened for another 6 months). The message is: longer periods leave developers too relaxed about the change.

3) Keep usage data of your system: By knowing which users are using which features, system maintainers can target those particular users via e-mail to remind them about upcoming changes. This approach has been studied in previous work [10] and it was also adopted by Facebook.

4) Blackout tests: Before taking the old versions offline permanently, try it for short periods of time. Twitter’s blackout tests approach has been successful in reminding developers that a change in the API is upcoming; the approach has also been appreciated by developers.

5) Provide an example of interaction with the API: Something not gathered directly from the analysis presented in this paper but rather collected from the API integration Pain Survey, is the developers' need for an (up-to-date!) example of how to interact with the API.

To summarize, web service APIs drive the evolution of software. Clients are forced to update by the API providers which contrasts with the statically linked libraries. However, in order to ease that evolution, we think the five aforementioned guidelines should be taken into account.

C. Threats to Validity

We now identify factors that may jeopardize the validity of our results and the actions we have taken or intend to take.

External validity. While we have quite some variety both in terms of API providers as well as in API client projects, it remains to be seen whether our observations still hold for (a) API providers who charge money for usage of the API, as they might be more reluctant when deprecating older version of the API which in turn might imply losing customers, and (b) for closed source API clients, whose developers might be inclined to upgrade quicker in order to satisfy their (paying) customer base with the latest security fixes and/or features. In future work, we will expand our investigation in this direction.

Construct validity. We have measured the impact of evolving APIs on clients by investigating the code churn. While code

---

churn is very valuable, it does not sufficiently take into account the relative complexity, nor the time needed to perform change tasks. In future work, through developer interviews we will investigate the actual effort cost of these maintenance tasks.

Reliability validity. There might be bias in the manual interpretation of the impact of change. To minimize bias the lead author who performed the investigation, thoroughly discussed all findings with the co-authors.

VI. RELATED WORK

Maintenance of service-based systems. Lewis and Smith were among the first to recognize that maintenance of service-based software systems is different from maintaining other types of software systems [11]. In particular, they highlight the importance of impact analysis for service providers as they have to consider a potentially unknown set of users.

Recent efforts in the area of program comprehension tools for service-based systems come from White et al., who look at both static and dynamic analysis to better understand these systems [12], and Espinha et al, who present their tool Serviz that allows to investigate how the topology of a service-based system can change at runtime [10] and how different users use a service-based system in different ways [8].

Evolution of APIs. Robillard and DeLine conducted a large-scale investigation among 440 profession developers at Microsoft to establish what makes APIs hard to learn [13]. Their observations are that the most severe obstacles developers face pertain to the documentation and other learning resources.

Dig and Johnson try to understand the nature of changes to APIs [5]. From the five case studies that they analyzed in detail, they found that over 80% of the API-breaking changes can be classified as being refactorings.

Henkel and Diwan present CatchUp!, a tool that captures API refactorings and subsequently replays these changes at the client side within the IDE [14].

Dagenais and Robillard present SemDiff, tool-support for recommending API-method replacements for methods that were broken during the evolution of the API [7].

The research presented above highlights some work in the area of understanding and evolving statically linked APIs. To the best of our knowledge, there is no prior work that describes how web service APIs evolve. An interesting non-peer reviewed work in this field is a survey conducted on the pains of API integration.

VI. CONCLUSION

In this paper we perform an exploratory study regarding the impact of web service API evolution. Our findings are that while API providers seem to agree on the technologies that implement their web service APIs, no consensus is reached regarding how and when to evolve them.

The impact of API evolution on the code of the clients is limited as the changes caused by the API evolution are around the average code churn for maintenance tasks and the changes are mostly confined to part of the system.

As the evolution is indeed inevitable, we also found that the different evolution policies impact the satisfaction of API client developers. To help mitigate this problem, we provide a list of recommendations such as not changing the API too often and performing blackout tests.

Future work. We aim to extend our investigation to a wider range of API providers and a larger selection of projects using these APIs. Additionally, we aim to analyze whether web service API changed impact open-source and closed-source applications differently. Do these closed-source projects apply more urgency to their changes due to their paying customers?

Finally, we also want to investigate whether the closed-source API providers’ policies differ from those of open-source APIs where client developers have no direct say in the evolution process.

ACKNOWLEDGMENTS

The authors would like to acknowledge NWO for sponsoring this research through the Jacquard ScaleItUp project.

REFERENCES

