How Does Web Service API Evolution Affect Clients? [1]

Jun Li[1,2], Yingfei Xiong[1,2], Xuanzhe Liu[1,2], Lu Zhang[1,2]
[1]Key Laboratory of High Confidence Software Technologies, Ministry of Education
[2]Software Engineering Institute, Peking University, Beijing, 100871, China
Email: {lijun09, xiongyf04, liuxzh, zhanglu}@sei.pku.edu.cn

***Abstract*—Like traditional local APIs, web service APIs (web APIs for short) evolve, bringing new and improved functionality as well as incompatibilities. Client programs have to be modified according to these changes in order to use the new APIs. Unlike client programs of a local API, which could continue to use the old API, clients of a web API often do not have the option not to upgrade, since the old version of the API may not be provided as a service anymore. Therefore, migrating clients of web APIs is a more critical task. Research has been done in the evolution of local APIs and different approaches have been proposed to support the migration of client applications. However, in practice, we seldom observe that web API providers release automated tools or services to assist the migration of client applications.**

In this paper, we report an empirical study on web API evolution to address this issue. We analyzed the evolution of five popular web APIs, in total 256 changed API elements, and carefully compared our results with existing empirical study on API evolution. Our findings are threefold: 1) We summarize the API changes into 16 change patterns, which provide grounded supports for future research; 2) We identify 6 completely new challenges in migrating web API clients, which do not exist in the migration of local API clients; 3) We also identify several unique characteristics in web API evolution.

***Keywords*—Software Engineering, Software Maintenance, Web Service API Evolution**

I. INTRODUCTION

With the popularity of service computing, web services are being published by companies and organizations. Client programs access these services via their APIs (called *web service APIs*, or *web APIs* for short), forming web applications. Today, many important applications in our daily life belong to this type of web applications. For example, Gmail is such a web application where the JavaScript client program invokes web APIs to get the mail data.

Ideally, the API elements (such as methods) of web services should not change, and the client and the server could evolve independently without affecting the other side. However, in reality, web APIs evolve due to various reasons, such as bringing new functionality and fixing bugs, and the client applications may have to be changed to adapt to the new APIs.

In traditional local applications, when some API elements (such as classes and methods) of a local API (an API without network interactions) change, the client may choose to continue using the old API if client developers do not want to upgrade. However, in the service paradigm, the service of an old API is often shut down after a certain period, and the client has to be upgraded to adapt to the new API, otherwise the client will stop working. Thus, dealing with API evolution should be a much more serious problem in web applications than local applications (desktop applications that only invoke local APIs).

Given the importance of migrating client programs, a lot of research has been devoted into this area and different approaches [5-11] have been proposed to automate the migration process. However, in practice, we seldom observe these approaches be adopted in the migration of web API clients. Most migrations are still performed manually, with no automated supports.

To support the migration of web API clients in practice, we believe that understanding how web APIs evolve is a necessity. On one hand, understanding web API evolution helps us identify the gap between existing approaches and the need of web application migrations, knowing where to put research efforts in. On the other hand, the characteristics of web API evolution could be useful in building effective approaches that make use of these characteristics.

Some researchers have realized the importance of understanding API evolution, and conducted systematic empirical studies on API evolution [1][2][3][4][13]. However, some of these studies focus on local APIs [1][2][4]. Other studies focus on automatic differing of WSDL interfaces [3][13]. We believe that studying the changes at WSDL level is not enough for guiding the migration of the clients. To really understand how web API evolution affects clients, we need an in-depth study of API changes on the semantic level.

In this paper we report our empirical study on web API migration, focusing on how API changes affect the clients. Our subjects are five popular web APIs, ranging from a personal calendar to twitter, from global service to local service in China. We carefully analyze their change history by reading the migrating guide, comparing reference documents, and experimenting with prototype clients, and identify in total 256 changed API elements that will cause incompatibilities. We then classify these changes and compare the results with existing empirical studies on local APIs. We also survey existing approaches to client migration and identify problems not considered in existing approaches. Web APIs are often provided in two levels: low level APIs as direct

[1] This work is supported by the National Natural Science Foundation of China under Grant No.61121063 and No.61202071, the High-Tech Research and Development Program of China under Grant No.2012AA011207, the National Basic Research Program of China (973) under Grant No. 2011CB302604, the Key Program of Ministry of Education, China under Grant No.313004 and NCET.

978-0-7695-5025-1/13 $26.00 © 2013 IEEE
DOI 10.1109/ICWS.2013.48

HTTP requests and high level APIs in popular programming languages such as Java and JavaScript. If no confusion will be caused, we may refer to both as API libraries. Our study focuses on the low level, while we also compare the results with the changes on the Java library for one of our subjects.

Our findings can be summarized as follows.

1) We summarize 16 patterns for web API evolution and count their frequency in the evolution. These data provide grounded guidance for designing approaches to migrating web API clients. (Section IV)
2) We identify 6 new problems in migrating web API clients. These problems are not considered by existing work or do not exist in local API evolution. (Section V)
3) We identify two unique characteristics of web API evolution. First, web APIs change much more frequently at the Java wrapper library level (short for *Java level*) than at the HTTP request level (short for *HTTP level*). Second, web API evolution reduces functionality much less often, but seems to affect more methods than local API upgrade. (Section VI)

We also compare our findings with existing empirical studies (Section VII).

In the rest of the paper, before jumping to our finding, we will first briefly introduce the background of web APIs in Section II and describe our experiment setup in Section III. We also discuss the threats to validity in Section VIII and conclude the paper in Section X.

II. Background: Web APIs and Wrapper Libraries

As mentioned before, web APIs are often provided in two levels, the core is at the HTTP level. In this level, the client accesses the services via direct HTTP requests and responses, and the formats of the requests and responses are defined in various protocols, such as SOAP, XML-RPC, and REST. A typical RESTful HTTP request is a GET request in the following format.

http://domain/methodname.format?access=&para1=&para2=&...&paraN=

domain: the address of server
methodname: name of the method
format: the format of return data
access: developer's unique 'Access Token'
para1…**paraN**: parameters of the method

When a server receives such a request, it performs the operations defined by the service, and then returns an HTTP response message, usually in JSON or XML. An HTTP request may also be protected under secure authorization, and has to be accessed via authorization protocols such as OAuth. In such a case, the client first obtains an access token and then sends the token along with the HTTP request to access the service.

Since HTTP level access requires many low level operations and the knowledge of various protocols, high-level libraries in popular languages are often provided. For example, Google provides its API in Java, Python, .Net, and many other languages. These libraries are usually provided as wrappers of APIs at the HTTP level, transforming the high-level method invocations into HTTP requests and parsing the HTTP response messages as in-memory objects.

III. Data Set

A. Web APIs

We choose five popular web services as our subjects: Google Calendar API[2], Google Gadgets API[3], Amazon Market Web Service[4] (Amazon MWS), Twitter API[5] and Sina Weibo API[6]. Google Calendar API provides access to Google Calendar, a popular online calendar service. It has three versions, and we choose the latest two versions (versions 2 and 3). Google Gadgets API allows the creation of Google Gadgets, and has only two versions (1 and 2). Amazon MWS facilitates the programming of data exchange, and has two versions (1 and 2). Twitter API allows the access of tweets and relationships among users, and has two versions (1.0 and 1.1). Sina Weibo API allows access to Sina Weibo data, a Twitter-like service in China, which is reported to have more messages published per second than Twitter. It has two versions (1 and 2)

Most of these APIs are in RESTful APIs, and services are provided in both HTTP level and in wrapper libraries using popular languages. One exception is Google Gadgets, where only a JavaScript library is provided, and many API methods execute locally within the browser. However, since the local executions still depend on the execution environment dynamically downloaded from Google, we still consider these methods web API methods. Another exception is Amazon MWS, which was provided as a SOAP service in version 1, and evolved into a RESTful API in version 2.

We choose these services according to four selection criteria. First, the API should have a large amount of clients; otherwise the API changes may be too random to be representative. Second, the services should cover different application areas. These APIs we chose cover tweet, calendar, online transaction, and gadgets in web. Third, the APIs should come from different companies and countries. Since API change is human behavior, we believe that the culture from different companies and different countries would influence the API evolution. The API libraries we chose are from four companies and two countries. Finally, the APIs must have good API reference documents and API migration guides, otherwise we need to find the semantics of all API elements before and after update using reverse engineering, which is infeasible given the resources we have.

[2] https://developers.google.com/google-apps/calendar/
[3] https://developers.google.com/gadgets/
[4] https://developer.amazonservices.com/index.html/
[5] https://dev.twitter.com/
[6] http://open.weibo.com/

B. Collecting the Changes

In order to obtain the accurate API evolution relations, we follow the steps listed below.

1) We downloaded API reference documents and API migration guide between two versions.

2) For each deprecated API element mentioned in the migration guide, we find the element and its replacement (if available) in both reference documents, and acquire the detailed changes between two elements by comapring the text description in the reference documents. If the text description is unclear, we write client programs to interact with the server to verify the functionality of these API elements.

3) When the migration guide could not include all changes or have some mistakes, we also compare the two reference documents directly to find whether there are more changes between the two versions. We perform this comparison by first matching API elements by name, then comparing the description text of matched elements, and finally checking all unmatched elements. By applying this step in Sina Weibo API, we found an omission and a mistake in the migration guide.

For all RESTful APIs, we perform the above process on the HTTP level. We also perform the above process on the Java wrapper library of Google Calendar API, in order to compare the changes on different levels. For Google Gadget, we perform the above process on the JavaScript library. For Amazon MWS, we ignore the protocol differences between SOAP and REST, and only focus on the essential difference between the abstract methods. At the HTTP level and the JavaScript level, the API elements we compare are mainly methods (as defined in Section II).

An API change is a specific change on part of an API element, such as HTTP request domain or the input parameters. API changes can be classified into breaking changes and non-breaking changes. Breaking changes are changes that will cause failures in client applications, including both compile-time failure and runtime failure. Changing the name of a method and changing preconditions of a method belong to this category. Non-breaking changes are changes that do not affect client applications. In other words, applications are compatible with new API. Adding a new method belongs to this category. In this paper we only consider breaking changes.

Table I summarizes the APIs and their changed elements. The first column presents the APIs we chose, and next column presents the versions of each API for comparing. We present the amount of total API elements of each API in the third column, the amount of changed API elements are presented in the fourth column. Last column presents the proportion of changed elements in total API. If there is an API change in one API element, we call this element changed API element. There can be more than one API change in a changed API element.

We can see in Table I, in average, more than half of the API elements become incompatible in the new version. This indicates a possible huge amount of work in migrating the client programs. This number is also quite distinct from an existing study on local APIs evolution [1][4], which shows only 30% of the API elements change in average in API evolution. We will discuss more about the discrepancies from existing studies on local API evolution in section VII

TABLE I. SUBJECTS

Projects	Versions	Total API elements	Changed API elements	Proportion
Google Calendar	version 2-3	47	38	80.1%
Google Gadgets	version 1-2	72	33	45.8%
Amazon MWS	version 1-2	31	21	67.7%
Twitter API	version 1-1.1	106	91	85.8%
Sina Weibo API	version 1-2	95	73	76.8%

IV. APIs CHANGE PATTERNS

In this section, we specify how API elements change. We further classify breaking changes into changes causing compile-time error and changes causing runtime-error. The former includes method signature changes and type changes in the high-level library, and the clients using this API will receive a compile-time error. At the HTTP level, since no compilation stage is involved, this means the changes on format of the HTTP requests and response messages, and an error code will be returned immediately if invoking the API with the old format. The second category means that the change will not have a visual effect at compile-time in the high-level library, but the client application may crash or misbehave after sending the HTTP request.

The classification of API changes is shown in Table II. Except the last three rows, the column headers are change patterns. The row headers are the web APIs. The cells are the number of the change APIs in the corresponding Web APIs. We summarize the API changes into 16 change patterns, where 12 of them are changes causing compile-time errors and 4 of them are changes causing runtime errors. In the following we discuss the change patterns one by one.

Following the work of Dig and Johnson [1][4], we also classify change patterns as refactorings and non-refactorings. Last three rows show the amount of refactorings and non-refactorings in all the breaking changes, and the proportions of refactorings. These numbers will be discussed in Section VII.

A. Changes Causing Compile-Time Errors

Add or Remove Parameter The number of parameters may be changed in API evolution. Usually, the reason of adding parameters is to enhance the ability of a method, e.g., querying more information from the database, while the reason of removing parameters is to weaken the ability of a method. Another reason of removing a parameter is that the parameter is no longer needed. For example, the provider of Sina Weibo API removes parameters of several methods during the upgrade, because these parameters are used in an old

security protocol that has been replaced by another one in the new version.

Both adding and removing parameters can cause problems in migration. When the removed parameters represent an important functionality, developers need to find the replacement of the missing functionality. When a parameter is added, developers need to decide the value for the parameter.

Change Type of Parameter This kind of change only appears in Amazon MWS API evolution, such as merging several independent simple parameters into one complex composite parameter.

For changes in this category, developers can easily migrate the client application in most cases. Usually, the new parameter can be synthesized by old parameters directly, and this process can be done automatically if developers know the synthesis rule from old parameters to new parameters.

Change Type of Return Value This pattern is similar to "Change Type of Parameter," and appears in Google API and Amazon API.

Delete Method For most cases, methods are deleted because their functionality is subsumed by other methods. However, there are cases where a method is removed and no replacement can be found. In the latter case, the migration of clients relying on the method becomes a big problem. This is also quite different from local API migration, as clients of local API could run with a copy of the old library and call the deleted method in the old library. We will discuss more on this issue in Section V.

An interesting issue we notice is that one method in Sina Weibo API version 1.0 is removed and the migration guide also claims the functionality of this method is removed and no replacement will be provided. However, actually we find that the functionality of this method can be achieved by combining the return value of several other methods which are not deleted. In other words, the functionality of this method is actually subsumed by other methods, but the API provider does not realize it.

Rename Method, **Rename Parameter** These two patterns are quite common in all APIs we surveyed. The main reason is to give self-explanatory names to methods and parameters. The migration for such changes is relatively easy comparing with other change patterns, and it can be fully automated with existing tools.

Change Format of Parameter, **Change Format of Return Value** These two patterns mean the type of a parameter or a return value does not change, but the format of them changes. For example, in Google Calendar API, a method accepts a string type parameter that needs to be encoded using URLEncode in the version 2.0. But in version 3.0, developers need only pass a string type parameter without encoding. These patterns are discovered in four APIs except Twitter API. In most cases, the goal of the changes is to facilitate the development of new client applications.

Migrating client programs can be partially automated in this category if the preprocessing step can be detected and removed.

Change XML Tag This pattern only occurs in Google Calendar API, where the value name used in the new JSON format differ from the original tags used in the XML messages. This pattern can be solved automatically.

Combine Methods This pattern means several methods are combined into one method. We find this pattern in the evolution of Amazon API. Amazon MWS API is a data-intensive API, there are a large amount of data need to be exchanged. In the old version of this API, developers need to interact with the server more than one times to get required data. But after evolution, developers can acquire the data from only one invocation. This decreases the latency in communication. However, the migration for this change is hardly automatable, since we need to find out possible consecutive invocations to the involved methods and group them into one.

Split Method We find this pattern in Amazon API evolution. It does not mean that one method split into several methods, but that one method was replaced by two different methods in different conditions. In Amazon API, the functionality of a method named 'PutInboundShipment' is to create or update an item about shipment in the database. If the item already exists, its record is updated; otherwise is created. In the new version, if the user wants to create an item, they should use the method named 'CreateInboundShipment', otherwise they should use 'UpdateInboundShipment' for updating the record of that item.

Expose Data This pattern appears in Google Calendar API, which provides data service. Data can be placed in a deep hierarchy, or can be organized in a flatter hierarchy. For example, in version 2.0 of Google Calendar API the resource path of 'originalEventId' and 'originalStartTime' are represented in the Atom format as follows:

```
<atom:entry>
<gd:originalEvent href="originalEventAtomId"
      id="originalEventId">
      <gd:when startTime="originalStartTime"/>
 </gd:originalEvent>
</atom:entry>
```

And in version 3.0 the resource path of these two methods represented as JSON format as follows:

```
{
    "recurringEventId": originalEventId,
   "originalStartTime": originalStartTime,
}
```

B. Changes Causing Runtime-Errors

Unsupport Request Method HTTP request methods include get, post, put and delete. In Sina Weibo API and Twitter API evolution, some methods in these APIs change from supporting two request methods in old version to supporting only one request method.If client applications just use a request mode that is supported by new version of an API, we need to do nothing. But if we use other unsupported request methods, we should modify the request method in our applications.

Change Default Value of Parameter This pattern appears in Sina Weibo API evolution. In an HTTP request,

TABLE II. API CHANGE PATTERNS

Type of change	Google Calendar (GC)	Google Gadgets （GG）	Amazon MWS (AM)	Twitter API (TA)	Sina Weibo (SW)
Decrease or Increase Number of Parameter	0	2	5	15	5
Change Type of Parameter	0	0	17	0	2
Change Type of Return Value	0	3	18	0	0
Delete Method	9	12	0	17	18
Rename Method	13	18	12	8	38
Change XML Tag	7	0	0	0	0
Rename Parameter	0	0	0	3	14
Change Format of Parameter	0	3	10	0	0
Change Format of Return Value	7	0	8	0	3
Combine Methods	0	0	1	0	0
Split Method	1	0	1	0	0
Expose Data	10	0	0	0	0
Unsupport Request Method	0	0	0	4	5
Change Default Value of Parameter	0	0	0	0	7
Change Upper Bound of Parameter	0	0	0	0	4
Restrict Access to API	0	0	0	0	3
Refactoring	29	35	52	43	76
Non-refactoring	18	3	20	4	23
Proportion	61.7%	92.1%	72.2%	91.5%	76.8%

parameters may have default values. Most of these parameters are about quantity, such as how many tweets can be displayed in one page. When the default values of such parameters change, we classify these changes in this category. The reason is that changes of default values may break the (potentially well-designed) layouts of a page, and such problems can only be captured at runtime.

Change Default Value of Parameter This pattern appears in Sina Weibo API evolution. In an HTTP request, parameters may have default values. Most of these parameters are about quantity, such as how many tweets can be displayed in one page. When the default values of such parameters change, we classify these changes in this category. The reason is that changes of default values may break the (potentially well-designed) layouts of a page, and such problems can only be captured at runtime.

To migrate the clients affected by this pattern, developers can explicitly specify the values of arguments when invoking the changed methods.

Change Upper Bound of Parameter This pattern only appears in Sina Weibo API evolution. Some particular parameters have upper bounds, e.g., a parameter may indicate how many tweets should be returned in one method invocation, and the upper bound is set for the maximum number of tweets that can be returned. When an upper bound becomes smaller, we classify this change as this pattern. Note that when an upper bound becomes larger, it is not a breaking change. Whether this pattern will cause problem in migration depends on the argument passed to the corresponding methods. If the argument is always smaller than the new upper bounds, nothing needs to be done for the migration. Otherwise, several invocations may be needed to retrieve all the needed data.

Restrict Access to API This pattern appears in Sina Weibo API evolution. Some methods are sensitive to information such as a method acquiring the private message of a person. So in new version of Sina Weibo API, the API providers improve the access authority of these methods. If developers want to access these methods, they should apply the authorization from API providers.

In this case, automatically migrating client applications is almost impossible, but a good migrating tool could provide useful help for developers.

TABLE III. GOOGLE CALENDAR EVOLUTION AT JAVA LEVEL

Type of Change	Frequency
Change Type of Return Value	7
Rename Class	6
Delete Class	10
Replace Class	3
Delete Method	4
Rename Method	20
Move Method	4
Expose Data	16

V. NEW CHALLENGES IN WEB API MIGRATION

There are a lot of approaches [5-12] to automating migration of clients for API evolution, and all we know are designed for local APIs. Although in principle these approaches should also work on web APIs, it is still an open question whether any new problems will emerge if they are actually applied. In this section, we try to partially answer this question by comparing the changes we discovered with the existing approaches, and checking whether there are any problems not considered in these approaches. Our findings are summarized as six challenges listed below.

1) *Transformation between JSON and XML:* Unlike local APIs, web APIs usually provide data services other than function services, so how to orgnize the data is an important issue. As mentiond before, the two most popular data formats are JSON and XML. JSON is a lightweight data format. It is usually much shorter than XML when desribing the same amount of data. In the web API evolution, many API providers replace the XML format with the JSON format. In such cases, the client applications also need to be updated to adapt to the new data format. The transformation between XML and JSON does not exsit in local API evolution, and thus is not considered by existing approaches.

2) *M to N Mapping:* Unlike local API, each invocation of a web API needs to access the remote web server, and the network latency will constitute a major part of the API invocation time. When mutiple API invocations are needed in one task, the accumulative latency will possibly become unacceptable. Acordingly, a typical type of API update is to merge several API methods into one big method, e.g., the "Combine Methods" pattern described in the previous section. However, as far as we know, all existing approaches that automatically migrate the client could only replace one method invocation into multiple method calls, but cannot replace multiple method invocations into one.

3) *Delete Method:* In local API evolution, if we need to use a method that exsits in the former version but removed in the new version, existing approaches either run both versions of the library together, or copy the code of former version to the client side [6]. But in web API evolution, the old version of the API will be shut down after a certain period, and developers have no way to access the deleted methods. The deletion of obsoleted methods is a new problem in migrating web API clients and new approaches need to be investigated.

4) *Authorization Protocol Change:* In the 1.0 version of Sina Weibo API, OAuth [7] 1.0 protocol for authorizing developers to access this API is used. In Sina Weibo API 2.0, the authorization protocol is also upgraded to OAuth 2.0. As a result, all API invocations on the client side need to be changed in accordance with the new version of the authorization protocol. All existing approaches we surveyed focus on changes on individual API methods, and could not handle such protocol change that affects all API invocations.

5) *Rate limit:* Responding an API invocation could be expensive. Since many services can be accessed for free, API providers often add a limit to the rate that the service can be invoked, and this limit can be changed from version to version. For example, Sina Weibo API 1.0 did not have a rate limit, but API 2.0 allows only 1000 invocations per hour. As a result, client programs that invokes the API too frequently need to be changed, by prefetching the needed data and/or pending the invocations into the server. Existing appraoches do not consider this problem because this problem does not exist in local API evolution.

6) *Authorization of API Access:* In Sina Weibo 1.0, any client applications can invoke any API methods. Realizing this actually opens doors to malicious applications, the providers of Sina Weibo 2.0 only allow authorized applications to access sensitive API methods. As a result, even if the client application has been authorized to access sensitive methods, all calls to those methods have to be changed to pass an additional access key. Existing approaches do not support changes at the authorization level as most local APIs do not involve authorization.

VI. CHARACTERISTICS OF WEB API MIGRATION

By comparing the changes of web APIs and local APIs, and between different levels of web APIs, we identify several unique characteristics of web API migration, as follows.

1) *Web APIs change more frequently at the HTTP level than wrapper library level.*

To compare the changes between wrapper library level and HTTP level, we summarize the API changes of Google Calendar API in Java level and the results are shown in TABLE III. An interesting observation is that there are many more API changes at the Java level than at the HTTP level—there are 70 changes at the Java level, but only 47 changes at the HTTP level. The main reason of this phenomenon is that input and output parameters in the HTTP level are organized into classes in the Java level, resulting a lot of changes in classes. First, there are purely refactoring changes on classes that have no counterparts on the HTTP level. Some changes in "Move Method", "Rename Class" and "Delete Class" patterns belong to this category. Second, one change on the parameters of an HTTP method may cause several changes on the classes. For example, a change in the "Expose Data" pattern often lifts a data item higher in the hierarchy in an HTTP input, but this may result in the change of several "getXXX" methods, and, sometimes, deletion and introduction of new methods and classes.

This finding indicates that developing a migration tool at HTTP level is potentially easier than at Java level. In addition, migrating clients at HTTP level is potentially more universal: wrapper libraries in different languages all invoke

[7] http://oauth.net/

the HTTP level API, and a migration tool at the HTTP level works for all wrapper libraries.

2) *More correct replacements can be eaiser found for web APIs than local APIs.*

Cossette and Walker's empirical study on local API [2] where one experiment is to identify replacements of incompatible API elements using six recommendation techniques, including reading release documents. The paper reports that, a single recommendation technique could only find in average 20% of cases, and there are 21.2% cases where no replacements can be found by any of the six techniques. Our result on web API is quite different. On our data set, we found replacements for the vast majority of the incompatible API methods by only reading the migration guides. In addition, for some methods that are claimed to be no longer supported in the migration documents, we still found that these methods can be simulated by combining the return values of several other methods. This indicates that web APIs have possibly better release documents.

3) *Web API evolution affects more methods than local API changes.*

Dig and Johnson's empirical study on local API [1][4] reports that no more than 30% of the API elements will change in one upgrade. However, in our study, as shown in Table I, at least half of the API elements are changed. This indicates that potentially web API evolution has a more widely impact on the applications than that of local API evolution. The reason for this discrepancy is not clear yet. Since the number of changed methods is affected by many issues such as the goal of the upgrade and the business decisions of the service provider, this discrepancy is also possible an occasional correlation due to the small number of upgrades in the data set, not a universal property of web API evolution.

VII. Related Work

As mentioned before, there are several studies on local API evolution [1][2][4]. We have already discussed two discrepancies between our findings and their results in Section VI. Here we discuss other important findings that are confirmed or contradicted in our work.

An important result in [1][4] is that more than 80% API changes are refactorings. This conclusion is verified in our work (see last three rows in Table II), we can see in average more than 80% API changes are refactorings in those APIs evolution. This result indicates that the tools solving refactoring in local API evolution can potentially be also applied in web API evolution. However, please also note that being refactorings does not mean that the migration is fully automable, as already pointed by Cossette and Walker [2].

Cossette and Walker [2] report several important findings in their study. Besides the "correct replacements are hard to be discovered" finding which we already discussed in Section VI, another important finding is that as low as 12.8% client applications influenced by API evolution can be automatically migrated from the old API to the new API. Our result presents some similarities from this finding. We classified the API changes on their automatability in a similar way in [2], and the details are described in Table IV. We found that the lowest proportion of automatically migrating web applications is 16.7%. This result indicates that automatically migrating client applications is a hard problem, even if at the HTTP level.

There are existing studies on the evolution of web APIs [3][13]. However, these studies explore how to automatically obtain the service changes by comparing WSDL definitions, focusing on the development history of web API rather than the effects on the clients. Furthermore, WSDL describes the syntax of web API methods. It is not enough to know the effect on the clients by looking at the syntax. For example, we cannot know precisely whether one method is a replacement of another method if we do not read the API documents and try out the methods with experiments. In addition, a lot of services nowadays are provided in RESTful style, where no WSDL is provided. Some change patterns are also summarized in the studies. However, these patterns are more coarse-gained architectural patterns compared to our fine-gained patterns. For example, one pattern is called "Aggressive Evolution", indicating a lot of elements are changed in one upgrade. Such pattern does not tell how the clients are affected, as an aggressive evolution could still be non-breaking changes.

Many efforts have been put in automating client migration in API evolution. In general, we can classify the migration into two steps: 1) find out how API changes; 2) modify the client applications accordingly. Existing approaches focus on automating the two steps.

Existing approaches that automate the first step can be classified into two categories. One is to record the changes by tools [5][6], called *operation-based* method [7]. Catchup! [5] and ReBA[6] provides an eclipse plugin to record the API changes when API providers update the library in eclipse. All the changes are represented as refactorings, and the process is automatic. The second way is to compute API changes by comparing the source code and documents of two releases [7][8][10][12][14].

Depending on the dependability of API changes, the approaches for the second step can also be classified into two categories. First, when the API changes are stable and precise, such as the recorded refactorings or change rules described by the API providers, the corresponding approaches [5][6][9][11] modify the client code automatically. Second, when the API changes are discovered by heuristic rules, the corresponding approaches [8][10] suggest updates on the client programs, and the developers will make the final decision based on the suggestions. Nevertheless, none of the approaches address the challenges discussed in Section V.

TABLE IV. Automatable on Migration

	GC	GG	AM	TA	SW
Fully Automatable	20	18	12	11	52
Others	27	20	60	36	47
Proportion	42.6%	47.4%	16.7%	23.4%	52.5%

VIII. THREATS TO VALIDITY

The main threat to external validity is that our study is based on a small data set, with five projects and 256 changed API elements. However, since we need to manually investigate API documents and conduct experiments with prototype code, it is very hard to increase the size of the data set. The data sets in existing empirical studies [1][2][4] reported in literature are also in comparable sizes. To alleviate this threat, we focused on increases the variety of projects, and our projects are chosen from different application domains and from companies with different cultures.

The main threat to internal validity is that our study heavily relies on the migration guide and reference documents, and these documents may contain mistakes. To see how large this threat was, we manually constructed a migration guide of Sina Weibo by only reading its reference documents, and then compared our migration guide with the official migration guide. As a result, we found one omission and one error in the migration guide. This shows that although errors do exist in the documents, their proportion may be small.

IX. CONCLUSION

In this paper, we have carefully analyzed five popular web service APIs, and draw some conclusions which are useful for application developers. First, we have summarized 16 patterns in web API evolution and given the frequency of each pattern in each web API evolution. This indicates web API evolves in limited patterns, and a tool addressing all these patterns could potentially automate the migration of clients. Second, we have described some new challenges that cannot be solved well by existing methods, and shown some unique characteristics in web API evolution. These challenges indicate where we should put research efforts in, and the characteristics could be helpful in attacking these challenges. Finally, we have discovered that some important conclusions derived from local API evolution also exist in web API evolution, e.g., more than 80% of API changes are refactoring, and some conclusions in local API no longer hold in web API evolution, e.g., when identifying the replacements of incompatible API elements, using migration guide could resolve the vast majority of the cases, rather than around 20% in local APIs. These findings could be useful to design migration tools.

REFERENCES

[1] D. Dig and R. E. Johnson, "The role of refactorings in api evolution," Proceedings of International Conference on Software Maintenance (ICSM05), pp. 389-398.

[2] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE12), no. 55.

[3] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An Empirical Study on Web Service Evolution," Proceedings of International Conference on Web Services (ICWS11), pp. 49-56.

[4] D. Dig and R. E. Johnson, "How do APIs evolve? A story of refactoring," Journal of Software Maintenance 18(2), 2006, pp. 83-107.

[5] J. Henkel and A. Diwan, "CatchUp!: Capturing and replaying refactorings to support API evolution," Proceedings of International Conference on Software Engineering (ICSE05), pp. 274-283.

[6] D. Dig, S. Negara, and R. Johnson, "ReBA: Refactoring-aware binary adaptation of evolving libraries," Proceedings of International Conference on Software Engineering (ICSE08), pp. 441-450.

[7] S. Meng, X. Wang, L. Zhang, and H. Mei, "A History-Based Matching Approach to Identification of Framework Evolution," Proceedings of International Conference on Software Engineering (ICSE12), pp. 353-362.

[8] B. Dagenais and M. Robillard, "Recommending adaptive changes for framework evolution," Proceedings of International Conference on Software Engineering (ICSE08), pp. 481-490.

[9] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," Proceedings of ACM SIGPLAN conference on Object—oriented programming systems languages and applications (OOPSLA05), pp. 265-279.

[10] H. A. Nguyen, T. T. Nguyen, G. W. Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," Proceedings of ACM SIGPLAN conference on Object—oriented programming systems languages and applications (OOPSLA10), pp. 302-321.

[11] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative APIs ," Proceedings of International Conference on Software Engineering (ICSE10), pp. 205-214.

[12] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," Proceedings of International Conference on Software Engineering (ICSE10), pp. 195-204.

[13] D. Romano and M. Pinzger, "Analyzing the Evolution of Web Services using Fine-Grained Changes," Proceedings of International Conference on Web Services (ICWS12), pp. 392-399.

[14] D. Dig, C Comertoglu, D. Marinov, and R. E. Johnson, "Automated detection of refactorings in evolving components," Proceedings of European Conference on Object-Oriented Programming (ECOOP06), pp. 404-428.