Unveiling the Mystery of API Evolution in Deep Learning Frameworks

A Case Study of Tensorflow 2

Zejun Zhang*, Yanming Yang†, Xin Xia†§, David Lo‡, Xiaoxue Ren*, John Grundy†
*College of Computer Science and Technology, Zhejiang University, China
†Faculty of Information Technology, Monash University, Australia
‡School of Information Systems, Singapore Management University, Singapore
{zejunzhang,xxren}@zju.edu.cn,{Yanming.Yang, Xin.Xia, John.Grundy}@monash.edu, davidlo@smu.edu.sg

***Abstract*—API developers have been working hard to evolve APIs to provide more simple, powerful, and robust API libraries. Although API evolution has been studied for multiple domains, such as Web and Android development, API evolution for *deep learning* frameworks has not yet been studied. It is not very clear how and why APIs evolve in deep learning frameworks, and yet these are being more and more heavily used in industry. To fill this gap, we conduct a large-scale and in-depth study on the API evolution of Tensorflow 2, which is currently the most popular deep learning framework. We first extract 6,329 API changes by mining API documentation of Tensorflow 2 across multiple versions and mapping API changes into functional categories on the Tensorflow 2 framework to analyze their API evolution trends. We then investigate the key reasons for API changes by referring to multiple information sources, e.g., API documentation, commits and StackOverflow. Finally, we compare API evolution in non-deep learning projects to that of Tensorflow 2, and identify some key implications for users, researchers, and API developers.**

***Index Terms*—API evolution, API documentation, Deep learning, Tensorflow 2**

I. Introduction

API libraries have always played an important role in efficiently writing programs. To provide simple, powerful, and robust API libraries, API developers continually attempt to evolve their APIs. Researchers have investigated API evolution in various domains, e.g., Web API [1], [2] and Java API [3]. Dig et al. [4] found that over 80% of breaking changes are refactorings, and suggested to develop refactoring-based migration tools. Brito et al. [3] found that 39% of breaking change candidates may affect clients, and the motivations are to support new features, simplify the APIs, and improve maintainability.

Currently, deep learning has attracted widespread attention and made major breakthroughs in many fields, e.g., computer vision [5], natural language processing [6] and software engineering [7], [8]. Several popular deep learning frameworks have emerged, and many versions have been released. However, API evolution in deep learning frameworks has not been investigated. There is no clear picture of how and why APIs evolve in deep learning frameworks.

§Corresponding author.

To fill this gap, we conduct a case study of API evolution on Tensorflow 2. Tensorflow is the most popular deep learning framework on Github [9]. Specifically, the number of Github projects using Tensorflow is 89,918 before 1th September, 2020, which is more than that of other deep learning frameworks (e.g., PyTorch [10] and Theano [11]), and the number of stars in Tensorflow is about 150, 000, which is the largest among all deep learning frameworks. Moreover, TensorFlow 2 is a major leap from the existing TensorFlow 1 with the following key differences: ease of use, eager execution, intuitive higher-level APIs, and flexible model building on any platform [12].

Prior studies on API evolution have three limitations. First, their research objectives are limited. Specifically, they do not consider many versions of API evolution [1], or they only focus on a specific category of API changes, e.g., deprecation and compatibility [13], [14]. Second, they omit some API changes, e.g., some studies mainly extract syntax of API changes without considering code changes with the same API signatures [15], [16], [1]. Other researchers extract API changes based on API artifacts, e.g., release notes, which only contains some API changes and some descriptions are too short to understand [4]. Finally, they ignore the reasons for API evolution but mainly focus on how API changes affect programs [16] and reflect on software artifacts [1].

To alleviate the above limitations, we propose to mine API changes based on API documentation of all versions on Tensorflow 2 to ensure consideration of more API changes than release notes and change logs. Also, API documentation contains rich information, e.g., API declaration and description of the function, parameter, and return value. Therefore, we can extract syntax API changes based on API declaration and more code changes from their natural language descriptions than prior studies. Moreover, Tensorflow API documentation offers the link to API source code. We can localize commits that are related to API changes, which can help us analyze the reason for API changes.

In this paper, we mine 6,329 API changes based on API documentation of all versions on Tensorflow 2. And then we associate these API changes with Tensorflow 2 framework

by classifying different modules into 6 functional categories with card sorting approach [17], which is because Tensorflow uses modules to aggregate APIs with similar functions. Then, we analyze reasons for API changes by referring to multiple information sources (e.g., commits and StackOverflow) and classify 10 reasons based on card sorting approach. Our study finds the number of API additions is increasing and the number of API deletions is decreasing with the update of API version. And the major API changes are related to low-level APIs of Tensorflow (e.g., data structures and error handling) [18] and model-related APIs (e.g., model training and evaluation) [19]. Furthermore, the main reasons for API changes are efficiency and compatibility.

We also discuss the difference between non-deep learning projects and deep learning framework-Tensorflow 2. Then, by analyzing the data distribution of the reasons for API changes, we find Tensorflow 2 makes changes to better support debugging, ease-to-use high-level APIs to quickly develop deep learning programs, and high-performance APIs to accelerate program runtime. Moreover, we analyze one potential problem and suggest that API developers consider migrating Keras into Tensorflow entirely in the future to prevent unexpected errors of "tf.keras" module. Finally, we find 3 common problems in API documentation and provide corresponding suggestions for researchers.

This work makes the following key contributions:

- To the best of our knowledge, we are the first to conduct a case study of API evolution in deep learning frameworks.
- We perform quantitative studies to classify API changes and investigate the reasons for API changes.
- We find differences between non-deep learning projects and a deep learning framework-Tensorflow 2, and provide practical implications for users, API developers, and researchers.

II. Motivation

Deep learning has made breakthroughs in many areas of software engineering, such as code search [8], defect prediction [20], and code migration [7]. However, it is difficult for developers to write an effective deep learning program from scratch. One reason is that deep learning requires programmers to have a certain mathematical foundation. Another is that deep learning programs involve multiple stages (e.g., data preparation, model setup and model training), a wide variety of layers (e.g., convolutional layer, pooling layer and fully connected layer), massive hyperparameters (e.g., learning rate and batch size), etc.. To overcome these difficulties, many deep learning frameworks provide high-level APIs, support distributed computing, and offer friendly debugging tools for users.

Tensorflow is a popular deep learning framework and contains substantial APIs to help users develop deep learning programs. Figure 1 presents the framework of Tensorflow 2 [1]. Core Tensorflow supports distributed running on multiple devices (e.g., CPUs, GPUs and TPUs). The next layer is the abstract layer, which provides components for users to build neural network models, and set metrics and losses for model evaluation. The third layer contains high-level APIs, e.g., keras and estimator, to help users construct and train models quickly. When there is a model, Tensorflow now standardizes the Saved Model, which can run on a variety of runtimes, e.g., cloud, web browsers ,and mobile devices. Besides, Tensorflow provides tensorboard to help users to analyze programs visually. For example, users can see metrics in real-time during training.

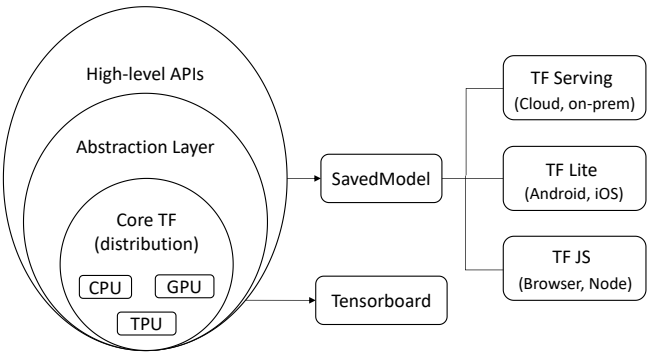

Fig. 1: Tensorflow Framework

Given its widespread popularity in industry, we wanted to answer the following two research questions about TensorFlow:

- **RQ1: How do API changes evolve on Tensorflow 2?**

Prior studies investigated the API evolution of non-deep learning projects. To understand the API evolution of deep learning framework-Tensorflow 2, we mine different API changes (e.g., API addition) and then map API changes into Tensorflow 2 architecture.

- **RQ2: What are reasons for API evolution on Tensorflow 2?**

Prior studies analyzed reasons for few API changes in non-deep learning projects. To analyze the reasons for many API changes in Tensorflow 2, we use card sorting approach to make classification. In this way, users can understand and use Tensorflow 2 better and help developers can evolve APIs better.

III. METHODOLOGY

A. Data Collection

Previous API evolution works mainly analyzed the syntax changes of API declaration, which omits code changes of APIs with the same API declaration. Since Tensorflow API documentation contains description of the function, parameters, and return value, we can find code changes by comparing the description between the two consecutive versions, such as changes in functions and types of parameters and return values. Particularly, Tensorflow has APIs available in several languages, e.g., Python and Java, but Python API is at present the most complete and the easiest to use [2]. Therefore, we

[1] http://tiny.cc/pt2zsz

[2] http://tiny.cc/emmzsz

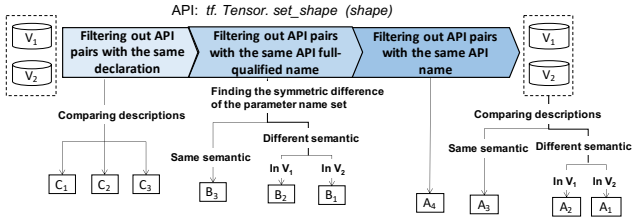

Fig. 2: The process of mining API changes

Note: $\mathbf{V_1}$ and $\mathbf{V_2}$ are two adjacent API versions, $\mathbf{A_1}$ is "API addition", $\mathbf{A_2}$ is "API removal", $\mathbf{A_3}$ is "API name's modification", $\mathbf{A_4}$ is "API move", $\mathbf{B_1}$ is "Parameter addition", $\mathbf{B_2}$ is "Parameter removal", $\mathbf{B_3}$ is "Parameter name's modification", $\mathbf{C_1}$ is "Parameter description's modification", $\mathbf{C_2}$ is "Method description's modification", $\mathbf{C_3}$ is "Return value's modification".

extract API changes based on Python API documentation of all versions (i.e., version 2.0 to 2.3) on Tensorflow 2 [21].

B. Mining API changes

By reading a range of API documentation, we find it mainly contains API declaration and description of methods, parameters and return values. Thus we focus on mining 10 types of API changes.

Figure 2 illustrates our API mining process. Below we explain concrete concepts using the "tf.Tensor.set_shape (shape)" API as an example. The API full-qualified name consists of module name, class name and API name, corresponding to "tf", "Tensor", "set_shape" respectively. API declaration consists of API full-qualified name and parameter lists.

We first mine descriptions of modifications to method, parameter, and return value, based on the difference of the corresponding descriptions with the same API declaration of two APIs in V_1 and V_2. This can reflect code changes with the same API declaration of two APIs. Next, we extract parameter-related changes by finding different parameter names with the same API full-qualified names in V_1 and V_2. We first compare their parameter descriptions in adjacent versions, if parameter descriptions are same, the corresponding parameter pairs are parameter names' modification. Otherwise, we determine the corresponding parameter category according to which version the parameter name appears in. Finally, for remaining APIs, we mine API name-related changes based on API name. Specifically, if there are API pairs with the same API names, they are API move. For remaining API-name related changes (i.e., API addition, API removal and API name's modification), the approach is the same as the approach of mining parameter-related changes. According to the process, we mine a total of 6,329 API changes.

C. Mapping API changes

After mining the API changes, we map them onto the Tensorflow 2 framework. Tensorflow uses modules to aggregate APIs with similar functions, so we can make functional categories based on all modules of Tensorflow 2. One author refered to Tensorflow 2 architecture in Figure 1 and topics of deep learning [19] to divide all modules into different categories. Another author then verifies the information and then provides suggestions for improvements. After incorporating suggestions, the two authors discuss to reach an agreement. We classify API changes into 6 functional categories in the Tensorflow 2 framework:

(1) Model-related changes: API changes that are related to deep learning that do not depend on any programming framework. Specifically, it includes the stacking of various layers, parameters initialization, neural network operations (loss, activation, and optimization function), training, testing, predicting, evaluating, loading and saving models (e.g., tf.keras and tf.nn) [19], [22].
(2) Data processing-related changes: API changes that involve data-related operations before the model training phase. It contains data reading and data preprocessing (e.g., tf.data and tf.io) [23].
(3) Framework-related changes: API changes are related to low-level APIs of the Tensorflow 2 framework. Different deep learning frameworks may design and implement different APIs, or users do not commonly use these APIs. However, these APIs are the foundation that users write and run deep learning programs. It contains the data type and structure, error handling, testing and debugging, basic configuration information, and raw mathematical calculation (e.g., tf and tf.math). [18], [21]
(4) Utility-related changes: API changes that are related to summarizing and analyzing deep learning programs (e.g., tf.summary and tf.profiler). [18]
(5) Distributed computing-related changes: API changes that are related to distributed computing from multiple CPUs, GPUs and TPUs (e.g., tf.distribute and tf.tpu) [24].
(6) Deployment-related changes: API changes that concern the deploying of Tensorflow models on mobile, embedded, and IoT devices in tf.lite module [25].

D. Identifying reasons for API changes

1) Information sources for change reasons: When analyzing API changes, we find it is not enough to obtain reasons only from API documentation. We identify the reasons for API changes from the following six information resources: Tensorflow API documentation, commits, issue reports, pull requests, Tensorflow community, and StackOverflow.

When extracting API changes from the previous version to the current version, we first read API documentation to analyze reasons. If the description is not enough to infer reasons, we use code links in the API documentation to localize commits related to API changes. As commit messages may mention issues or pull requests, we also refer to the information to analyze reasons. If the information is still not enough to give the reason, we manually find related questions in StackOverflow or Tensorflow community [26].

2) Reason categories for API changes: Since there are no predefined reason categories for API evolution, we use a card sorting approach [17] to identify the reason types of API changes. There are two iterations and we used two evaluators.

TABLE I: An example of the reason for an API change

API change	tf.linalg.matvec Method description change: 2.0: ...we **must have** shape(b) = shape(a)[:-2] + [shape(a)[-1]]... 2.1: ..., and shape(a)[:-2] able to broadcast with shape(b)[:-1]...
A1 annotation	correct false constraint information that doc is inconsistent with code
A2 annotation	"must have" information is wrong because 2.0 version supports broadcast

Iteration 1: In the first iteration, we randomly sample 362 API changes [3] to make reason categories. There are two steps:

Step 1 Annotation of reason: Two authors separately annotate the reason with a short description. For example, Table I shows the two authors' annotations about changes of the method description of "tf.linalg.matvec" API from version 2.0 to 2.1. Then, two authors discuss the disagreements to reach an agreement.

Step 2 Reason category for API changes: They work together to group all annotations, and then give the two types of information for each group: category and corresponding definition, which are inspired by the quality attributes of software systems [27].

Iteration 2: In the second iteration, two of the authors independently annotate API changes with reason categories. If there are no existing categories that can cover new API changes, they annotate the API changes with reason annotations. After gathering all reason annotations, they find that there are no new categories. We use Cohen's Kappa measure [28] to examine the agreement between the two authors. The Kappa value is 0.87, which indicates a high agreement between two authors. Then, two authors discuss the disagreements to reach an agreement.

IV. RESULTS

A. RQ1 Classification of API Changes

Table II shows the numbers of the 10 types of API changes for two consecutive versions. There are 6,329 API changes in total from version 2.0~2.3. As the version updates, the total number of API changes increases, the number of API additions increases, the number of API removals descreases, and the number of other types of API changes fluctuates.

We explain notable API changes in all versions. API addition accounts for 39% of changes from version 2.1~2.2. By examining the API changes, we find many APIs were added into the "tf.raws" module that contains raw operations used by Tensorflow library writers. This accounts for 58% of changes from version 2.2~2.3 because a third-party library "NumPy" is integrated into the Tensorflow framework as "Tnp". API removal accounts for a large portion from version 2.0~2.1 because developers hide many APIs from the documentation. There are no API name modifications, which is reasonable because the change can affect users' usages. The proportion of API move is generally relatively small. We find that most of APIs are moved from experimental APIs into stable APIs, e.g., "tf.random.experimental.get_global_generator" is moved to "tf.random.get_global_generator". The number of Parameter removal from version 2.1~2.2 is larger than other phases, because developers fold Keras into Tensorflow, which also leads to an increase in Parameter addition from version 2.1~2.2, e.g., "*args" parameter of "tf.keras.applications.Xception" is replaced with parameters of the same API of Keras. The number of Parameter addition from version 2.1~2.2 is larger than other phases because developers make an extension for old APIs and the reason above for folding Keras into Tensorflow. Besides, Parameter name's modification accounts for a low percentage. The percentage of Parameter description's modification from version 2.1~2.2 is larger than other phases because developers modify and supplement description in Dataset and Keras. The distribution of Method description modification in multiple phases is even. The percentage of return value's description from version 2.1~2.2 is also higher than other phases because developers supplement the description in Keras and Estimator.

[3]the sample size are statistically significant with a confidence level of 95% and a confidence interval of 5

TABLE II: The number of Tensorflow 2 API changes from version 2.0 to 2.3

Version	A_1	A_2	A_3	A_4	B_1	B_2	B_3	C_1	C_2	C_3	Total
2.0~2.1	142	200	0	21	23	8	0	101	186	50	731
2.1~2.2	1489	16	0	20	276	94	0	508	176	136	2715
2.2~2.3	2222	10	0	10	83	8	4	327	164	55	2883

TABLE III: The numbers of functional categories of API changes based on Tensorflow 2 framework from version 2.0 to 2.3

Version	Model	Data	Utility	Distribute	Tensor	Deployment	Total
2.0~2.1	348	186	1	22	174	1	732
2.1~2.2	686	325	9	152	1543	0	2715
2.2~2.3	267	259	7	108	2240	1	2882
Total	1301	770	17	282	3957	2	6329

Table III depicts the numbers of distributions of the functional category of API changes based on the Tensorflow framework from version 2.0 to 2.3. "Model-related", "Data processing-related" and "Framework-related" APIs are constantly evolved in all versions, account for 21%, 12% and 63% respectively; "Distributed computing-related", "Utility-related" and "Deployment-related" API changes account for less than 5%. Specifically, "Model-related" changes are focus on high-level APIs in "keras" and "estimator" modules, which involve various operations such as components for building neural networks, training, testing, predicting, and evaluating models. And the portion of "Model-related" API changes in 2.1~2.2 is higher than other phases because developers mainly improve APIs of existing models (e.g., resnet); "Data processing-related" changes focus on data preprocessing such as data transformation, shuffling, splitting, and augmentation; "Distributed computing-related" changes mainly focus on supporting distributed running on TPUs; "Framework-related" changes are mainly low-level APIs for developers and numerical computation; "Utility-related" changes mainly focus on helping users analyze programs visually; "Deployment-related" changes are only two about mobile devices and embedded devices.

By analyzing all API changes, we find Tensorflow API developers are creating a more friendly framework for users

and developers. Specifically, to improve usability, they continually improve high-level APIs (e.g., Keras and Estimator); To provide convenience for developers, they group low-level APIs into specific modules such as "tf.raw_ops". In API evolution, they avoid making API changes (e.g. API name's modification and move) that can affect users in different versions, and continually improve the documentation quality.

B. RQ2 Reasons for API Changes

Table IV presents the distribution of reasons for different API change categories. "Efficiency" and "Compatibility" account for a large portion about 54%; "Convenience" and "Robustness" only account for less than 1%, other reasons account for 44%. We explain the 10 reasons for Tensorflow API evolution in detail.

1. Efficiency: API changes are for accelerating program runtime or saving resources. Developers mainly introduce third-party libraries or support distributed running to improve efficiency. The major functional category of the reason is framework-related change.

When users write deep learning programs, they usually need to use computation-related APIs such as customizing loss functions [29] or layers of neural networks [30]. Although users can use APIs of "Numpy" (np) that is a popular and strong computation tool such as "np.random" and "np.linspace", the efficiency is limited for deep learning programs. Since deep learning needs massive data, calculation and iterations, it takes a longer time than traditional programs. To solve the problem, developers integrate many APIs of "np" into Tensorflow as "Tensorflow NumPy" (Tnp). "Tnp" performs many compiler optimization and uses highly optimized Tensorflow kernels [31], which can make programs that use "np" run faster.

Google develops Tensor Processing Unit (TPU) to accelerate machine learning workloads for Tensorflow [32]. To support distributed running on TPUs, developers continually develop related APIs from Tensorflow API version 2.0 to 2.3. Specifically, developers add initialiation-related classes such as "TPUClusterResolver" to connect a remote cluster and initialize TPUs. And then they add "TPUStrategy" class to distribute data on TPUs. Besides, developers add many APIs to support custom distributed training. For example, users can use "TPUStrategy.run" API to define their own functions [4]. Finally, developers make existing functions run on TPUs. For example, they add "TPUEmbedding" class to support large embedding that turns data into vectors on TPUs.

2. Convenience: API changes are for helping programmers analyze, configure or debug programs. The major categories of the reason are framework-related and utility-related changes.

If deep learning programs produce problems such as low performance and wrong or poor results, it is more difficult for developers to analyze deep learning programs than traditional programs because deep learning programs have lots of layers, neurons and parameters. To overcome the difficulty, developers continually add a series of APIs to show various program states in a visualization way. For example, developers add the "tf.profiler.Trace" class to show a timeline of durations of Tensorflow operations or functions, and which part of the system executed an operations. In this way, users can easily observe problematic operations or functions that influence performance, and then solve problems faster.

Developers develop a number of compilers and optimizers to improve performance, so they need to add many APIs to help users conveniently configure programs, or get the configure information. For example, they add "is_built_with_xla" API in version 2.2 to help users know whether the program uses accelerated linear algebra (XLA) to speed linear algebra computation. Besides, to bridge heterogeneous ecosystem, they develop a multi-level intermediate representation (MLIR), which leads to the emergence of new APIs of supporting MLIR optimization. For example, developers add "enable_mlir_graph_optimization" API in version 2.3 to turn on MLIR function.

Tensorflow at first requires users to build a graph and then to create one session to execute programs. It is very inconvenient especially when users write a big model and need to directly get results of programs without creating sessions. To solve the problem, developers introduce eager execution that is an imperative environment to evaluate operations immediately. So, users can directly get results and use Python debugging tools to debug programs. The new mode makes developers add new APIs to support the eager execution of features. For example, developers add "tf.config.run_functions_eagerly" API in version 2.3 to make users use "tf.function" in an eager execution instead of a graph function.

3. Powerfulness: API changes are for enriching APIs with new functionalities to meet the different needs of developing programs. The major categories of the reason are framework-related, model-related and data-related changes.

API developers continually add attributes to existing classes such as various data structures and mathematical functions. For example, developers at first do not contain the length attribute in "tf.data.Dataset" class. As a result, many users propose that how to get the dataset size [5] and then developers add "tf.data.Dataset.__len__" API in version 2.3. Actually, the attribute is common to many classes of object-oriented languages (e.g., Java classes). However, Tensorflow API developers initially ignore the attribute. The similar circumstances that classes lack important attributes are common in Tensorflow, i.e., Tensorflow also adds getting eigenvalues and condition number in the matrix computation. On the other hand, API developers add a variety of functionalities to meet the different needs of users. For example, deep learning requires lots of data sets, but large size of data is not easy to come by. To allievate the problem, developers add many image preprocessing methods to augment datasets. Also, initialization

[4]http://tiny.cc/gt2zsz

[5]http://tiny.cc/pl2zsz

TABLE IV: The numbers of reasons for API changes within different functional categories

	Efficiency	Convenience	Powerfulness	Robustness	Expandability	Compatibility	Completeness	Conciseness	Correctness	Friendliness
Model	1	2	116	7	75	26	390	219	349	103
Data processing	28	0	81	7	82	81	277	72	136	6
Utility	0	10	3	0	13	0	2	0	2	0
Distribution	79	1	0	1	31	30	47	10	83	0
Framework	1872	20	247	3	76	1318	169	6	212	34
Deployment	0	0	0	0	1	0	1	0	0	0
Total	1980	33	447	18	278	1455	886	307	782	143

of weights can affect performace, so developers add many random functions for users to choose.

Although there are APIs to help users implement certain functions, sometimes users need easier APIs to implement the same functions. For example, initializing weights are important parts of deep learning. Before version 2.3, if users want to initialize the weights with one external model, they need to use the constructor to initialize these parameters, which is troublesome and error-prone. Therefore, developers add the "from_config" API in version 2.3 for users to instantiate weights by importing the configuration of models directly.

Users not only need high-level APIs to build models quickly, but also need to customize functions. Therefore, developers add many APIs to support flexibility. For example, developers add "tf.keras.Sequential.train_step" API in version 2.2 to support custom many training logics, e.g., metrics updates and loss computation.

4. Robustness: API changes are for handling various wrong input or negative circumstances. The major categories of the reason are model-related and data-related changes.

Deep learning programs usually have many hyperparameters (e.g., mean value in normal distribution), but users sometimes only need or know a few hyperparameters. To solve the problem, developers set parameters to default values as much as possible. Therefore, the program can still run normally when users ignore unfamiliar parameters. For example, in version 2.2, developers set default value of "padded_shape" parameter in "tf.data.Dataset.padded_batch" API as "None". When users do not set the parameter, programs will automatically pad all dimensions of all components to the maximum size. On the other hand, users know there are errors in deep learning programs, but they expect programs to ignore errors and continue running. Therefore, developers add parameters for users to select whether skipping errors. For example, when users use "tf.keras.Sequential.load_weights" API to load weights and the shape of some layers' weights is not consistent with their models, they want to skip these mismatched layers. Hence, users can set "skip_mismatch" parameter added in version 2.1 to make sure the program can run.

When training models, optimization may take a long time to find a satisfactory result. If the program is interrupted, the model needs to be retrained, which will waste a lot of time. Therefore, developers need to backup some important and necessary information. For example, developers add "BackupAndRestore" class in version 2.3 to make models restored to a previous checkpoint from interruptions.

5. Expandability: API changes are for making extensions based on the original API code instead of writing APIs from scratch. The major difference between powefulness and expandability is that powerfullness leads to the emergence of new features, which does not depend on the original APIs, while expandability is to make extensions to original APIs. We find developers make non-functional extensions and functional extensions.

For the non-functional extension, on the one hand, the emergence of new technologies (e.g., distribution and eager execution) has led to API changes. For example, distributed computing support data run and stored on many devices, so users need to specify devices to load data. Under the circumstances, developers generally add parameters into original APIs to support distributed setting. For example, developers add "option" parameter for the "tf.keras.Sequential.load_weights" API in version 2.3. With the "option", users can specify a device they want to load model weights. On the other hand, it is for expanding the original APIs' function scope, e.g., supporting more data types or operations on multiple dimensions. In this case, developers generally do not change API declaration, but they need to add the related description of the method, parameter, and return value to clarify code changes. For example, in version 2.3, the "x" parameter in "tf.test.compute_gradient" API also supports tuple data type, so developers add the parameter's description for the information.

For the functional extension, although developers provide certain functions, the options of function are limited to meet the needs of users, e.g., not supporting adjusting learning rate, explicit padding and grouped convolution. An example is that users have the requirements of grouped convolution for convolution operations because it can reduce computation cost [33] and also help people easily migrate from Caffe to Tensorflow [6]. Therefore, developers add "groups" parameter into "tf.keras.layers.Conv3D" API to support the function.

6. Compatibility: API changes are for supporting different platforms, systems, devices, and API versions. The major categories of the reason are framework-related and data-related changes. We find there are mainly backward compatibility and forward compatibility.

Backward compatibility is to make changed APIs in new versions that could normally run on old versions. When developers make API changes in API evolution, they generally consider whether changed APIs can run on the old version. Otherwise, users have to manually modify the old version's code to adapt to the new version, which is time-consuming and laboursome. For example, when developers add one "tf.data.TFRecordDataset.__len__" API, they also add

[6]http://tiny.cc/xl2zsz

one "__len__" API in "tf.compat.v1.data.TFRecordDataset" class so that users can use the new API in previous versions.

Forward compatibility is to make APIs in the current version can also run on future versions. In this way, users and developers can directly reuse these APIs in the future. For example, developers add the lowest APIs into "tf.raw_ops" module that never change semantic. Such stable APIs are convenient for developers to develop library APIs. And separately adding the module can make users automatically enjoy the benefits of optimized APIs. For example, both "tf.math" and "tf.raw_ops" contains the "Add" API. Since the module will not change the semantic of APIs, so developers add "tf.raw_ops.AddV2" API to make optimization. If they do not add the module but directly add the API in "tf.math" module, users who use the "Add" API cannot automatically enjoy the optimization benefits.

7. Completeness: API changes are for supplementing missing or important information. Without the information, users cannot obtain the information from documentation because previous documentation does not contain the information. The major categories of the reason are model-related, data-related and framework-related changes. We find developers mainly supplement three types of information in API documentation.

Developers usually supplement information about specific values of parameters. First, they supplement parameters' default values in the parameter description. Second, for parameters with a fixed set of options, developers clarify all options. For example, developers add the description-"supports batch_shape and batch_input_shape" for "**kwargs" parameter of "tf.keras.Input" API in version 2.1. In this way, users can see options parameters support to prevent from using non-existent options. The last is that they specify specific values and corresponding meaning for parameters with a group of constant values. For example, the "value" parameter of "tf.keras.backend.learning_phase_scope" API only has 0 and 1. Developers supplement description-"0=test, 1=train" to make users know the specific meaning of parameter values.

Developers often supplement function-related information to make users clearly know the function of APIs. Developers usually explain the specific behavior of APIs on specific cases. For example, for "tf.Tensor.__div__" API in version 2.3, developers add description-"if x and y are both integers, then the result will be an integer", to stress it is integer division. On the other hand, developers add additional notes to prevent users from inefficient or wrong usages of APIs. For example, "tf.keras.layers.Lambda" API can make users write arbitrary function as a layer. In version 2.1, developers add the description-"while it is possible to use Variables with Lambda layers, this practice is discouraged as it can easily lead to bugs", to advise users not to use the API but to write a subclass layer for complex and state computation to avoid bugs [7]

Developers also add other important information, e.g., supplementing important attributes and constraints of values,

[7]http://tiny.cc/8s2zsz

```
- from keras_applications import resnet
...
- @keras_modules_injection
- def ResNet101(*args, **kwargs):
-   return resnet.ResNet101(*args, **kwargs)
+ def ResNet101(include_top=True,..., classes=1000):
+ """Instantiates the ResNet101 architecture."""
```

Fig. 3: API changes for resnet API

value range and datatypes parameters support, purposes and benefits of APIs, and variables' changes APIs can bring. For example, developers add description-"this must be a floating point type" for "t" parameter of "tf.clip_by_norm" API in version 2.3 to clarify the constraints.

8. Conciseness: API changes are for removing redundant or useless information, or modifying tedious information. The major category of the reason is model-related change. We find developers mainly integrate Keras to Tensorflow and deprecate some APIs.

Previous Tensorflow API developers write implementation by mixing up Tensorflow and Keras. Now, they are replacing Keras with the "tf.keras" module, which can simplify the complexity of code. For example, Figure 3 describes that developers change the way of keras_modules_injection decorator into implementing detailed code such as specifying parameters of "resnet101" in version 2.2[8] because it is unnecessary to share the implementation in Keras and "tf.keras" module.

We observe that developers generally deprecate some APIs to recommend users to use other existing APIs because other APIs also implement the same or similar functions. Such API changes can remove redundant APIs and help users prevent confusion about similar APIs. For example, the "fit" API and the "fit_generator" API are both to train models. Users previously raised confusion about the difference between the two APIs on StackOverflow [9] and found the difference is small-the "fit" API does not support generator but "fit_generator" supports. To avoid the confusion, developers made "fit" API support generator. Then, developers deprecated "fit_generator" API in version 2.1 to keep simplicity.

9. Correctness: API changes are for fixing errors, modifying unreasonable API information, or updating obsolete information. The major categories of the reason are model-related, framework-related and data-related changes. We find developers mainly use three ways to correct mistakes.

Developers can wrongly expose some APIs in the documentation. On the one hand, these APIs are outdated APIs. For example, "tf.distribute.StrategyExtended.non_slot_devices" API is hid in Tensorflow 2.3 because it is only a Tensorflow 1 API. On the other hand, developers wrongly expose some APIs of subclasses from the base class. For example, "tf.keras.callbacks.ReduceLROnPlateau.on_batch_end" API is inherited from the "tf.keras.callbacks.Callback" class, but it is exposed to version 2.0. So, developers hide the API from the documentation.

[8]http://tiny.cc/bs2zsz
[9]http://tiny.cc/gs2zsz

In API evolution, developers find they wrongly place APIs into improper modules, so they move them into proper modules. They generally check based on whether the API in the module is functionally similar to many APIs on another module. For example, the "tf.image.encode_png" API is moved to the "tf.io" module because the module contains substantial encoding APIs. On the other hand, when developers integrate APIs of the third-party libraries into Tensorflow, they check whether the current APIs conform to norms of APIs of third-party libraries. For instance, Keras has various initializer classes, and Tensorflow developers reimplement them due to technical details. They at first placed them into other modules, which make these classes disconnect with Keras. To correct the error, they move initializer classes into the "tf.keras.initializer" module.

We find developers often provide wrong information in description of methods, parameters and return values. One particularly frequent error is about the description of data type. Developers at first roughly describe data type in API documentation, and then they begin precisely describe the information. For example, "x" parameter description about data type in "tf.Variable.__and__" API from version 2.2 is "Tensor", developers correct it as "tf.Tensor" in version 2.3. On the other hand, the original data type is outdated. For example, in version 2.2, the "iterator" parameter type of "tf.data.experimental.get_next_as_optional" API is "tf.data.DataSet", but it is updated to "tf.data.iterator" in version 2.3. So, the parameter description is be modified accordingly. Besides, outdated information also exists in function description. For example, developers modify method description of "tf.linalg.matvec" API-"...shape(a)[:-2] able to broadcast with shape(b)[:-1]" into "tf.linalg.matvec..." in version 2.1 because the broadcast mechanism has be supported very early, but developers forget to update the information in documentation.

10. Friendliness: API changes are for programmers to understand better, remember, and use APIs by adding auxiliary information. Specifically, documentation does not lack the information, but adding information can help users understand better. The major categories of the reason are model-related and framework-related changes. We observe developers mainly supplement two types of reference information to help users further understand the function and usage of APIs.

The first is to add code examples with run results[10]. In this way, users can intuitively know how to use these APIs and can easily use the code examples by copying and pasting. The second is to add links of paper, Wiki and other APIs to help users further understand the APIs in depth. For example, developers add links of other APIs into the description of "tf.ones" API because they represent similar functions [11]. In this way, users can not only employ previous experience to understand the API and also use the link to understand more similar APIs.

[10]http://tiny.cc/ms2zsz

[11]http://tiny.cc/qs2zsz

V. Discussion

A. Comparison With Non-Deep learning projects

Our study analyzes API evolution in the Tensorflow 2 framework by mining different types of API changes and analyze their reasons. Prior studies investigate API evolution on non-deep learning projects from their perspectives. So we inspect their differences and similarities between deep learning frameworks and non-deep learning projects.

For breaking changes that break backward compatibility (i.e., API removal and move, parameter addition, and removal), non-deep learning projects account for about 28% [34], Tensorflow 2 frameworks account for about 12%, which shows Tensorflow API developers pay more attention to avoid making breaking changes.

For reasons for API changes, Rediana et al. [1] and Brito et al. [3] investigated reasons for API evolution on DHIS2, which is an open source and web-based health management system, and on Java libraries and frameworks respectively. We find that the reasons they raised do not involve convenience and compatibility. Specifically, for convenience, since deep learning programs have lots of iterations, layers, and parameters, and their results are uncertain, it is difficult to find problems by only observing a program point. For example, when users check whether deep learning programs run normally, one important step is to monitor changing trends of accuracy and loss instead of checking a single value. To allievate the problem, Tensorflow continually includes a series of APIs to help users debug and analyze programs by showing program states visually. For compatibility, since Tensorflow 2 is a major leap from Tensorflow 1, it continually adds many APIs to help programs using Tensorflow 1 run on Tensorflow 2. Particularly, we find Tensorflow 2 pays more attention to efficiency than non-deep learning projects, especially in accelerating numerical computing and distributed computing.

B. Implications

Through the analysis of API evolution on Tensorflow 2, we found many of its advantages and its improvements to previous weaknesses, which is very friendly to Tensorflow 2 users. We find that the number of Tensorflow 2 API changes due to convenience is relatively small, not only because visual debugging is a difficult point, but also because its debugging tools are relatively good. More importantly, it has alleviated some debugging challenges, e.g., providing Debugger V2 GUI to help users work through bugs involving NaNs [12] proposed by Zhang et al. [35]. Also, for efficiency, Tensorflow 2 supports Numpy-related functions to be dispatched on GPUs and distributed computing on TPUs, which can accelerate program runtime. Besides, users can easily upgrade code from Tensorflow 1 to 2 because developers have made many API changes for compatibility. Finally, we find Tensorflow 2 continually improves ease of use in developing deep learning programs by making many API changes in high-level APIs (e.g., introduced Keras into "tf.keras" and independently

[12]http://tiny.cc/v38zsz

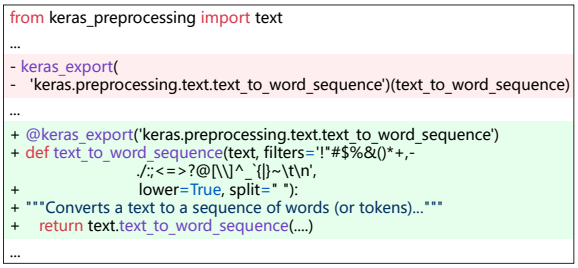

Fig. 4: Unexpected errors during migrating Keras into Tensorflow [13]

developt "tf.estimator") due to powerfulness, expandability, robustness, conciseness, correctness, and completeness, which is a weak point proposed by Nguyen et al. [36].

For Tensorflow API developers, considering correctness, we suggest they could consider reducing the dependency on Keras as much as possible by migrating Keras to Tensorflow entirely in the future. Specifically, Keras provides a series of high-level APIs, which is very convenient for users to construct and run neural networks quickly. So, developers currently introduces Keras into "tf.keras". To reduce the workload, they often depend on APIs of Keras, which can lead to unexpected errors. For example, Tensorflow contributors add comments for "tf.keras.preprocessing.text.text_to_word_sequence" API. Then, they submit a piece of code in Figure 4. Unexpectedly, the code cannot be successfully built because the "text" parameter conflicts with "from keras_preprocessing import text". Namely, when executing the return code, program mistakenly calls the "text" parameter instead of the "keras_processing.text" class. To eliminate the error, contributors temporarily modify the "text" into "input_text". Actually, reducing the dependency on Keras and migrating Keras into Tensorflow entirely can prevent many similar problems from occurring in the future.

For researchers, considering correctness, we suggest they could consider providing a tool to automatically detect and generate data type information in the description of API documentation such as parameter description. Since we notice API documentation frequently exists wrong or outdated descriptions about data type, developers need to manually correct the information, which is time-consuming, laboursome and error-prone. **Considering friendliness, researchers could consider providing a tool to automatically recommend code examples to API developers because huge APIs and a few API developers can lead to lagging effects and uneven quality of code examples.** For example, API developers add a piece of code example for "tf.keras.backend.random_uniform" API, which comes from users' requirements [14]. Although the code example provides the API attributes, it lacks real scenarios of API usages. Actually, the above API can be used in data preprocessing such as rotating images randomly by generating random numbers from Github [15]. **Besides, API developers could consider supplementing API version information in API documentation for each API to help users analyze programs.** For example, deep learning users ask how to implement a function and then use a piece of code from StackOverflow, sometimes the code can not run successfully because some APIs are not available on the current Tensorflow version.

[13]http://tiny.cc/xrszsz
[14]http://tiny.cc/6t2zsz
[15]http://tiny.cc/bt2zsz

C. Threats to Validity

Internal validity. The one is related to errors of our code of mining 10 types of API changes. To alleviate the problem, we manually check all API changes. The second is that card sorting of classification for API changes and corresponding reasons is subjected to subjective bias. To allievate the problem, two authors independently examine their results at least twice and annotate important information for verification.

External validity. It is related to the generalization of the functional category of API changes and corresponding reason categories. Our study is conducted on one deep learning framework-Tensorflow. The classification maybe not be applied to other popular frameworks. However, deep learning frameworks contain a lot of commonality [37], and we conduct a case study for 6,329 API changes. Therefore, we think that the classification can be adapted to other deep learning frameworks with some revisions.

VI. Related Work

API evolution has been studied in multiple domains. Danny Dig et al. [4] manually analyzed API changes from software artifacts, such as release notes and change logs, for four frameworks and one library. They found more than 80% of breaking changes are refactorings. Kim et al. [38] analyzed the relationship between API refactorings and bug fixes on Eclipse JDT. They found there is a growing number of bug fixes after API-level refactorings. Tyler et al. [39] analyzed the pace of Android API changes and adopting new APIs by users. They found adopting new versions needs a longer time than evolving APIs, and using new APIs to write programs is more error-prone than programs without API usage adaptation. Wei et al. [16] studied API changes and usages together on Apache and Eclipse Ecosystems. They found missing classes and methods occur more often and missing interfaces happen rarely. Laerte et al. [34] extracted syntax of API changes between two versions of many Java libraries. They found the number of breaking changes increases in API evolution, and few client applications are affected by breaking changes. Aline et al. [3] extracted API changes from 400 popular Java libraries and frameworks. They found API changes are mainly for new features and improving maintainability. Rediana et al. [1] extracted 38 API changes of Web API between version 2.26 and 2.27. They found these changes do not completely reflect on software artifacts, e.g., release notes and issues. From the above empirical studies on API evolution, we can see current studies focus on non-deep learning projects and do not conduct a large-scale and fine-grained study on reasons of API evolution.

VII. Conclusion and Future Work

In this paper, we conduct a case study on Tensorflow 2 API evolution based on API documentation. By mining 6, 329 API changes, we find the number of API addition increases, and the number of API removal decreases with the update of versions. And then we map these API changes into 6 functional categories based on Tensorflow 2. The results show that model-related, distribution-related, and low-level Tensorflow APIs are changed the most. Then, we categorize reasons for API changes into 10 reason categories and find the main reasons for API changes are efficiency and compatibility. Finally, we compare the difference between non-deep learning frameworks and Tensorflow 2 on API evolution. By analyzing Tensorflow 2 API evolution, we find Tensorflow 2 makes changes to better support high-level APIs, friendly debugging, and high-performance APIs for users to develop deep learning programs. Also, we recommend Tensorflow API developers considering migrating Keras into Tensorflow entirely in the future and give researchers new suggestions about improving API documentation. In the future, we plan to investigate more deep learning frameworks, such as PyTorch and Theano, to analyze their commonality and difference in API evolution. We expect to help users better choose deep learning frameworks, help API developers evolve efficiently, and inspire researchers to reduce API evolution's workload on deep learning.

Acknowledgment

This research was partially supported by the National Key R&D Program of China (No. 2019YFB1600700), Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021), ARC Laureate Fellowship funding scheme (FL190100035), ARC Discovery grant (DP200100020), and the National Research Foundation, Singapore under its Industry Alignment Fund Prepositioning (IAFPP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló, "Classification of changes in api evolution," in *EDOC*, 2019.

[2] S. Sohan, C. Anslow, and F. Maurer, "A case study of web api evolution," in *ICWS*, 2015.

[3] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and how java developers break apis," in *SANER*, 2018.

[4] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, 2006.

[5] A. Ioannidou, E. Chatzilari, S. Nikolopoulos, and I. Kompatsiaris, "Deep learning advances in computer vision with 3d data: A survey," *CSUR*, 2017.

[6] L. Deng and Y. Liu, *Deep learning in natural language processing*. Springer, 2018.

[7] Z. Zhang, M. Pan, T. Zhang, X. Zhou, and X. Li, "Deep-diving into documentation to develop improved java-to-swift api mapping," in *ICPC*, 2020.

[8] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *ICSE*, 2018.

[9] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *ISSTA*, 2018.

[10] N. Ketkar, "Introduction to pytorch," in *Deep learning with python*, 2017.

[11] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," in *Proc. 9th Python in Science Conf*, 2010.

[12] G. Developers, "Tensorflow guide," https://www.tensorflow.org/guide, 2020, accessed Aug. 4, 2020.

[13] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Cda: Characterising deprecated android apis," *EMSE*, 2020.

[14] H. Cai, Z. Zhang, L. Li, and X. Fu, "A large-scale study of application incompatibilities in android," in *ISSTA*, 2019.

[15] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP*, 2006.

[16] W. Wu, F. Khomh, B. Adams, Y.-G. GuÃ©hÃ©neuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *EMSE*, 2015.

[17] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[19] J. Han, E. Shihab, Z. Wan, S. Deng, and X. Xia, "What do programmers discuss about deep learning frameworks," *EMSE*, 2020.

[20] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *QRS*, 2015.

[21] G. Developers, "Tensorflow api documentation," https://www.tensorflow.org/versions, 2020, accessed Aug. 4, 2020.

[22] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, 2015.

[23] S. García, J. Luengo, and F. Herrera, *Data preprocessing in data mining*. Springer, 2015.

[24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[25] L. Shuangfeng, "Tensorflow lite: On-device machine learning framework," *JCRD*, 2020.

[26] G. Developers, "Tensorflow contributor community," https://www.tensorflow.org/community, 2020, accessed Aug. 4, 2020.

[27] L. Chen, M. Ali Babar, and B. Nuseibeh, "Characterizing architecturally significant requirements," *IEEE Software*.

[28] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica: Biochemia medica*, 2012.

[29] T.-M. Li, M. Aittala, F. Durand, and J. Lehtinen, "Differentiable monte carlo ray tracing through edge sampling," *TOG*, 2018.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[31] (2020) tensorflow numpy. [Online]. Available: https://www.tensorflow.org/guide/tf_numpy

[32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[33] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *CVPR*, 2018.

[34] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *SANER*, 2017.

[35] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *ISSRE*, 2019.

[36] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. L. García, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, 2019.

[37] A. Parvat, J. Chavan, S. Kadam, S. Dev, and V. Pathak, "A survey of deep-learning frameworks," in *ICISC*, 2017.

[38] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *ICSE*, 2011.

[39] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *ICSME*, 2013.