A3: Assisting Android API Migrations Using Code Examples

Maxime Lamothe, *Member, IEEE,* Weiyi Shang, *Member, IEEE,* Tse-Hsun (Peter) Chen

Abstract—The fast-paced evolution of Android APIs has posed a challenging task for Android app developers. To leverage Androids frequently released APIs, developers must often spend considerable effort on API migrations. Prior research and Android official documentation typically provide enough information to guide developers in identifying the API calls that must be migrated and the corresponding API calls in an updated version of Android (*what* to migrate). However, API migration remains a challenging task since developers lack the knowledge of *how* to migrate the API calls. There exist code examples, such as Google Samples, that illustrate the usage of APIs. We posit that by analyzing the changes of API usage in code examples, we can learn API migration patterns to assist developers with API Migrations.

In this paper, we propose an approach that learns API migration patterns from code examples, applies these patterns to the source code of Android apps for API migration, and presents the results to users as potential migration solutions. To evaluate our approach, we migrate API calls in open source Android apps by learning API migration patterns from code examples. We find that our approach can successfully learn API migration patterns and provide API migration assistance in 71 out of 80 cases. Our approach can either migrate API calls with little to no extra modifications needed or provide guidance to assist with the migrations. Through a user study, we find that adopting our approach can reduce the time spent on migrating APIs, on average, by 29%. Moreover, our interviews with app developers highlight the benefits of our approach when seeking API migrations. Our approach demonstrates the value of leveraging the knowledge contained in software repositories to facilitate API migrations.

Index Terms—API, software quality, mining software repositories, empirical software engineering

✦

1 INTRODUCTION

Software maintenance is one of the most expensive activities in the software development process [1]. To reduce the maintenance cost, developers often rely on reusing available software. The reused software helps abstract the underlying implementation details and can be integrated into innumerable software projects. In particular, calling application programming interfaces (APIs) is a common software reuse technique used by developers [2], [3]. These APIs provide well-defined programming interfaces that allow their users to obtain desired functionality without forfeiting development time.

However, in today's fast-paced development, APIs are evolving frequently. The Android API is one such example of a rapidly-evolving and widely used API [4], [5]. Prior studies [5], [6] found that Android is evolving at an average rate of 115 API updates per month. Such evolution may entail arbitrary release schedules and API deprecation durations and may involve removing functionality without prior warning [7]. Therefore, users must regularly study the changes to existing APIs and decide whether they need to migrate their code to adopt the changes. In fact, a prior study shows that developers are slower at migrating API calls than the API evolution speed itself [5]. As a consequence, there is fragmentation in the user base and slow adopters who miss out on new features and fixes [5].

Existing migration recommendation techniques [8]–[11] typically focus on identifying *what* is the replacement of a deprecated API (e.g., one should now be using methodB

instead of methodA), instead of *how* to migrate the API calls for the replacement (e.g., how to change the existing code to call methodB). However, a recent experience report shows that all too often, Android API official documentation clearly states *what* to replace for a deprecated API, while actually performing API migrations is still challenging and error prone [6].

There exist many publicly-available code examples online illustrating API usages. As an example, Google provides a set of sample Android projects on the Google Samples repository [12]. Developers often study these sample projects and other code examples (e.g., code from open source Android apps) to help them with API migration [2], [13]–[15]. Nevertheless, studying the code examples to know the changes needed for API migrations is a manual and time-consuming process. Furthermore, identifying *where* and *how* to apply migration changes puts an extra burden on developers during software maintenance.

In this paper, we propose an approach, named A3, that mines and leverages source code examples to assist developers with API migration. We focus on Android API migrations, due to Android's wide adoption and fast evolution [5]. Our approach automatically learns the API migration patterns from code examples taken from available code repositories, thereby providing varied example patterns. Afterwards, our approach matches the learned API migration patterns to the source code of the Android apps to identify API migration candidates. If migration candidates are identified, we apply the learned migration patterns to the source code of Android apps, and provide the resulting migration to developers as a potential migration solution.

To evaluate our approach, first A3 learns Android API

- *Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.*
  E-mail: (max_lam, shang, peterc)@encs.concordia.ca

```
  view.setFocusableInTouchMode(true);
− view.setBackgroundColor(mFrag.getresources().getColor(R.color.default_background));
+ view.setBackgroundColor(mFrag.getresources().getColor(R.color.default_background, null));
  view.setTextColor(Color.WHITE);
```

Fig. 1. An API migration example of the *getColor* API, in the *GridItemPresenter* class of *androidtv-Leanback* project in commit *6a96ad5*.

migration patterns from three sources of code examples: 1) official Android code examples provided by Google Samples [12], 2) migration patterns that are learned from the development history of open source Android projects i.e., FDroid [16] and 3) API migration examples that are manually produced by users. Our approach then applies Android API migrations to open source Android apps from FDroid and we leverage their test suites and manually run the apps to validate the correctness of the migration. Furthermore, we compared our approach to LASE [17], a tool meant to apply code edits learned from examples. Moreover, we carry out a user study to determine the actual and perceived usefulness of our approach. In particular, we answer three research questions.

RQ1 *Can we identify API migration patterns from public code examples?*
Our approach can automatically identify 80 migration patterns with 96.7% precision in Android APIs used in public code examples, and obtains a recall of 97% using our seeded repository.

RQ2 *To what extent can our approach provide assistance when migrating APIs?*
Based on 80 migrations candidates in 32 open source apps, our approach can generate 14 faultless migrations, 21 migrations with minor code changes, and 36 migrations with useful guidance to developers. Furthermore, interviews with four developers highlight a positive developer response to our migration examples.

RQ3 *How much time can our approach save when migrating APIs?*
Through a user study with 15 participants and 6 API migration examples, we show that our approach provides, on average, a 29% migration time improvement and is seen as useful by developers.

Previous research has proposed approaches such as Sydit [18] and Lase [17] to help developers with API migration; however, these approaches must be manually pointed towards pre-migrated examples without the ability to automatically retrieve or identify them [19]. Furthermore, the effectiveness of code examples on migration is affected by the context of the examples, whereby examples with closer contexts will waste less developer time when testing extraneous cases [19]. Therefore, by considering multiple examples from different contexts our approach generates well-fitted migration solutions.

Our approach can be adopted by Android app developers to reduce their API migration efforts to cope with the fast evolution of Android APIs. Our approach also exposes the value of learning the knowledge that resides in rich code examples to assist in the various tasks of API related software maintenance.

Paper Organization. Section 2 provides a real-life example of an API migration to motivate this study. Section 3 describes our automated approach, A3, that assists in Android API migration. Section 4 presents the design of the experiments used to evaluate our approach and Section 5 presents the results of our experiments. Section 6 provides a short survey of related work. Section 7 describes threats to the validity of this study. Finally, Section 8 concludes the paper.

2 A MOTIVATING EXAMPLE

In this section, we present an example, which motivates our approach based on learning migration patterns from code examples to assist in API migration.

In Android API version 23, the *Resources.getColor* API (as shown in Listing 1) was deprecated and replaced (as shown in Listing 2). In fact, the deprecation and replacement (*what* to migrate) are clearly shown in the official Android documentation [20]. However, even with the help from the documentation, the addition of a new parameter provides new challenges to the Android app developers. In particular, developers may not have the knowledge necessary to retrieve the *Theme* information nor to initialize a new object for the *Theme* to make the new API call. Moreover, since the old API call does not require any *Theme*, even if developers can provide a *Theme*, there is no information on how to preserve backward compatibility.

On the other hand, there exists open source example projects on Google Samples [12], i.e., *androidtv-Leanback*, a project on Github which presents several uses of the *Resources.getColor* method. With the introduction of a new Android API version, these code examples are also updated. By looking at an example, we find that it clearly demonstrates how to call the *Resources.getColor* API (see Figure 1). From the changes to the code example, we can see that to maintain backward compatibility, developers can simply pass a *null* object to the API. Without such an example, figuring out that a *null* value preserves backward compatibility would require trial and error from developers. By learning such a migration pattern in the code examples, the effort of the challenging task of *how* to migrate an API call can be reduced for developers.

Listing 1. Resources.getColor API before migration

```
public class Resources extends Object {
    public int getColor(int id)
}
```

Listing 2. Resources.getColor API after migration

```
public class Resources extends Object {
    public int getColor(int id, Resources.Theme
        theme)
```

However, finding these migrations on Google Samples is a laborious endeavor. First of all, the code examples do not index their usage of APIs. Developers may need to search for the API usage of interest from a large amount of source code. Second, even with the API usage in the code examples, developers need to go through the code history to understand the API migration pattern, i.e., *how* to apply these changes, which can be much more complex than the aforementioned example. Finally, even with the migration pattern, developers need to learn how to apply the migration on their own source code, which can be a challenging task [6].

Taking the aforementioned API *Resources.getColor* as an example, we can detect a total of 1,626 places where the *Resources.getColor* Android API is called in its deprecated form on a sample of 1,860 open source Android apps from FDroid [16]. Migrating *Resources.getColor* to Android API version 23 or later for all of those apps requires a significant amount of effort. Automating the above-described migration approach and providing the information of the learned pattern to developers can significantly reduce the required migration effort.

In the next section, we present our approach, named A3, which provides assistance in Android API migration.

3 APPROACH

In this section, we present an approach, A3, that assists in API migration by learning how to migrate API calls from code examples. Our approach consists of two steps: 1) identifying API migration patterns from code examples and 2) applying the migration pattern that is learned from the example to the source code of the Android apps. An overview of our approach is shown in Figure 2 and an implementation of our approach is publicly available online[1].

3.1 Learning API migration patterns from code examples

In the first step of our approach, we mine readily available code examples to learn API migration patterns. Such code examples can be found through online code repositories such as GitHub and FDroid [16]. Our approach also supports self-made examples, which allows users to produce their own migrated stubs, or use their own projects as data to feed forward into other projects.

3.1.1 Extracting APIs

The lack of uniformity between API documentation and source code presents a challenge to API migration approaches. There is currently no easy way to obtain a verified list of all migrated API methods for all versions of the Android API. Furthermore, the sheer size of the Android API presents a challenge to API migration. Manually searching

source code for all deprecated, removed, and modified APIs is an arduous task.

Hence, we automatically extract all the Android APIs for every Android version that is available from the official Android API documentation. However, the Android documentation may have discrepancies to the APIs in their source code[2]. For every version of Android, we obtain the Android source code and parse the source code using Eclipse JDT AST parser [21]. We identify a list of all public methods as available APIs.

3.1.2 Searching API calls from code examples

Algorithm 1: Our algorithm for searching API migration examples.

Input : Code example repository $exRepo$ and available API list $apiList$
Output: Migration example
1 /* First do basic lexical matching */
2 **foreach** api in $apiList$ **do**
3     $apiCallData \leftarrow api.\text{get}(callData)$
4     $apiParams \leftarrow apiCallData.\text{getParamCount}()$
5     $apiName \leftarrow apiCallData.\text{getName}()$
6     $potentialMigrations \leftarrow null$
7   **foreach** $commit$ in $exRepo$ **do**
8        $potentialMigrations \leftarrow$ $commit.\text{getAPIcalls}(apiName, apiParams)$
9   **end**
10   /* Selective check for all potential migrations */
11   **foreach** $call$ in $potentialMigrations$ **do**
12       $callAST \leftarrow call.\text{buildAST}()$
13   **if** $callAST.\text{matches}(apiCallData)$ **then**
14   saveExample($call$)
15   **end**
16 **end**

In this step, we identify the API calls that are available in the code examples. A pseudocode algorithm of this step is presented in Algorithm 1. One may design an approach that builds AST for all available code examples as well as the targeted Android app source code. Afterwards, one may use the ASTs to match the API calls in the source code to the code examples. However, such an approach would be time consuming due to the fact that 1) building ASTs is a time-consuming endeavor, 2) the complexity of AST matching is non-negligible and 3) the number of Android APIs that are available is large.

In order to reduce the time needed to identify API calls, and thereby reduce the challenges faced by developers who seek to migrate APIs, we first use basic lexical matching to limit the scope of needed AST building and matching. In particular, we first search all the files in the code example for strings that match API names. This technique allows us to quickly get a preliminary list of potential API calls. These files are then selected for further processing as potential matches. Although the basic lexical matching can lead to

(a) Step 1: Learning API migration patterns from code examples

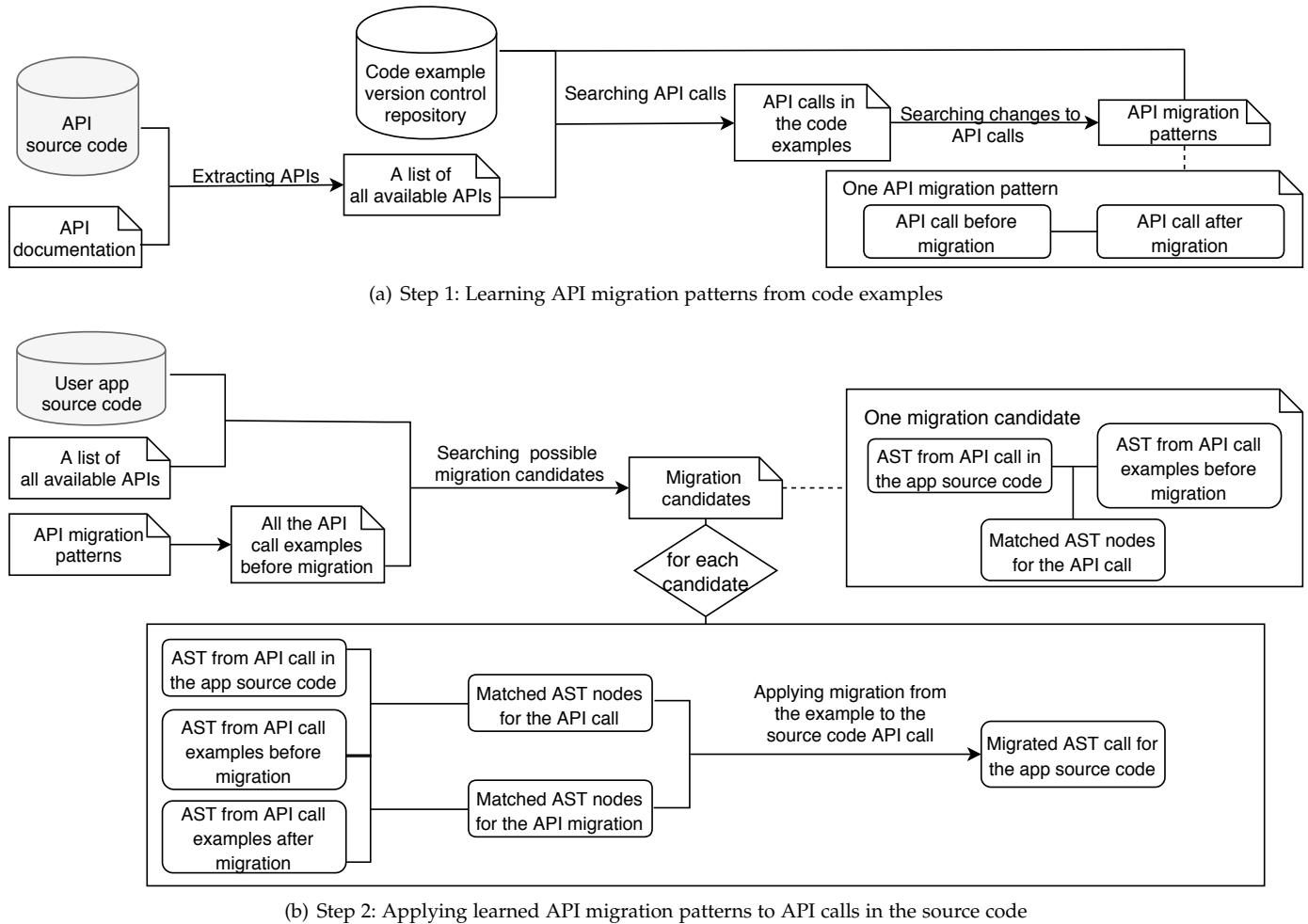(b) Step 2: Applying learned API migration patterns to API calls in the source code

Fig. 2. An overview of our approach, named A3, that assists in API migration

false positives, the goal is merely scoping down the search space and the following AST matching can remove these false positives. For example, in Figure 1 we specifically search for the keyword *getColor*, although it is possible for this basic search to falsely identify a similar post-migration API as a migration target, once the refined check with AST matching is complete i.e. these *getColor* false positives would be filtered out if they did not have exactly one integer parameter, the right return type, and all other discernible AST properties for *getColor*.

After the basic lexical matching, we apply AST matching on the files with potential matches. This allows us to scope down the previously identified results to obtain high precision findings without sacrificing too much performance. We leverage JDT [21] to parse the source code in the code examples to generate their corresponding ASTs. For each method invocation in the AST, we compare with the APIs that are potentially matched from the lexical search. If the AST can be fully built, we aim to obtain the perfect matches between method invocation and API declaration. In particular, a perfect match requires matching the method invocation name to an API declaration as well as having perfectly matched types for all parameters. In some cases, the code examples may not be fully complete (e.g., missing some external source code files or dependencies), leading to partially built ASTs. With the partially built file level ASTs, we consider

a match if there exist the correct import statement of the API, the correct method name, and the correct number of parameters. All method instances are saved, along with their invocation string. This data can quickly be reviewed by a developer to determine if a false positive was detected, making our approach transparent and understandable. Developers interviewed in RQ2 presented in Section 5, were presented and reviewed the migration instances that were related to their familiar app and confirmed our results.

3.1.3 Learning API migration patterns

In this step, we search API calls for every version of the code examples. For instance, if code examples such as Google Samples [12], or FDroid projects [16] are hosted in version control repositories, we detect API calls in every commit of the repository.

Searching every commit for a multitude of projects requires surmounting the challenge imposed by the large scale of available data from API user projects. We surmount the challenge imposed by the large amount of data available by leveraging the basic *diff* in the version control system (like Git) to determine in which commits a specific string is modified to collect commits that contain changes to API calls.

Commit level migrations allow us to reduce the amount of code modifications that could obfuscate an API migra-

tion. If we reduce the search scope to a granularity coarser than commit level we might miss certain migrations. Similar to the lexical search from the last step, the use of a diff tool is merely scoping down our search space. For the commits that potentially impact API calls, we parse the AST of the changed files in the commit. We compare the ASTs generated from the source code before and after the commit. If the API invocations in the AST are modified in the commit, we consider the commit as a potential API migration pattern. Hence, each API migration pattern consists of an AST built from the example before the migration and an AST built after the migration.

For the code examples that are hosted as text files outside of repositories, we apply a text diff on each two consecutive versions of the example instead of using commits to scope down the search space. The secondary AST matching step on these text files is identical to that from the version control repositories.

3.2 Applying learned API migration patterns to API calls in the source code

In our second step, we collect all the API migration patterns (i.e., ASTs built from the example before and after the migration) from the first step and try to apply these patterns to the API calls in the Android app source code.

3.2.1 Searching possible migration candidates

Similar to our first step, to resolve the challenges imposed by the scale of the problem at hand, we reduce the search space, our approach first uses API names to lexically search for API calls with available migration patterns in the source code of the targeted Android app.

Since migrations can be dependent on the context of surrounding code, we cannot assume that one migration example will suit all possible use cases. For example, API such as *Resouces.getColor(int)* can be present in a variety of use cases. To match valid migration patterns, we must not only match API calls, but also determine if an example matches a user's usage of the API. We leverage data-flow graphs to match the API calls in the migration patterns and the targeted Android app sources. We construct a data-flow graph from the API call example "before" the migration in the migration patterns. We also construct data-flow graphs in the API calls in the Android app source code. Only if the data-flow graph from the example is a subgraph of a potential API call in the Android app source code do we then consider this a migration candidate. This allows us to assume that the example API being used as a migration pattern has a similar use case to the API call in the targeted Android app.

3.2.2 Applying the migration from the example to API calls in the source code

Existing API migrations can contain implementation details that cause incompatibilities with other projects. Therefore, our approach must mitigate the challenges imposed by implementation details and allow developer interaction. We employ data-flow graphs, rely on the large number of migration examples provided by open-source projects to

---

Algorithm 2: Our algorithm for migrating code.

Input : Migration mapping $mappedDFG$ and client data-flow graphs $clientDFG$

Output: Migrated data-flow graph

1 /* $Traverse\ all\ data - flow\ graphs$ */
2 **foreach** $DFG\ in\ clientDFG$ **do**
3     $DFGMap \leftarrow mappedDFG$.get($DFG$)
4     $changedAPIs \leftarrow DFGMap$.getChangedAPI()
5   /* $Migrate\ all\ migrateable\ APIs$ */
6   **foreach** $changedAPI\ in\ changedAPIs$ **do**
7   changedNodes $\leftarrow$ $DFGMap$.getDataLinks($changedAPI$)
8   missingNodes $\leftarrow$ $DFGMap$.getNodesToAdd($changedAPI$)
9   /* $Adjust\ the\ data - flow\ graph$ */
10   DFG.addNewNodes(missingNodes)
11   DFG.migrateDFG(migrateableNodes)
12   **end**
13 **end**

---

obtain our migration examples, and let the developer have final say at every step of the approach.

If any migration candidate is found, we then attempt the migration. We first compute the migration mappings by comparing the examples before and after API migration. This mapping contains any changes that must be made to existing code statements, obtained by comparing the names and types of each code statement in the data-flow graph. The migration mapping also contains any new code statements that are present in the "after" migration example but were missing in the "before" migration example.

In order to obtain an accurate migration mapping between the "before" and "after" examples (i.e., changes that were made to migrate the API), we need to eliminate the changes on AST that are not related to the API migration. We achieve this by relying on the data-flow graphs that are built from the examples. We first remove all the nodes in the data-flow graph where all the associated nodes are perfectly matched between the "before" and "after" examples. Since they are perfect copies of one another, those nodes cannot contribute to a migration. We keep the nodes in the data-flow graphs that remain unmatched and are associated with the node that is of interest to the API call. Finally, we compare the nodes that are kept to find the matched data-flow graph for the API call in the Android app after migration.

Once we obtain the most likely migrated data-flow graph in the "after" API migration example, we produce a backward slice of the data-flow graph starting from the API call. In other words, we only look at nodes that give data to the API call. Based on our sliced data-flow graph, we then map each node in the "before" example to the most closely matched node in the "after" example. Any unmatched data linked nodes are considered to be new nodes and are saved to be added during migration.

Finally, we use the migration mapping to transform the project source code into the "after" API migration example. Pseudocode of our migration algorithm can be found in Algorithm 2. The transformation also looks for any object types that are matches between the "before" example and

the project source code to infer the names used in the Android app source code, to prevents the introduction of new variable names. Our approach can produce migrations that are interprocedural and intraprocedural. However, we limit the migration scope of our approach to single files since prior work has found that field and method changes account for more than 80% of migration cases [22]. Adding migration across multiple files would increase the chance of making mistakes, both when mining and applying migration patterns, and only account for a minority of migration cases.

To better explain our approach, Figure 3 illustrates an example of applying a migration from an example to an API call. The *before* and *after* migration examples illustrate potential example code obtained from sample projects. We can see that methods which are exact matches are seen as irrelevant since they add no information to the migration. Similarly, we can see that nodes after the migration node of interest, in this case a return statement, are also seen as irrelevant nodes. This is because it does not matter what the user does with their code after the API call (i.e., not related to the usage of the API). Other nodes are then matched between *before* and *after* examples to determine which node is most likely the migration. Any nodes that do not obtain a match are considered new nodes required for migration and will be added in user projects, as denoted by the color blue in our example. Once a match is produced between migration examples, the migration mapping can be applied to the user example, as presented by the arrows in Figure 3 and as explained by the pseudocode in Algorithm 2.

4 EVALUATION DESIGN

To determine the extent to which our approach, A3, can be used to assist with API migration, we conducted a series of experiments. We first present the research questions which we use to evaluate our approach. Then, we present our data acquisition approach for Android apps and code examples.

4.1 Research questions

We aim to evaluate our approach by answering three research questions. We present our research questions and their motivations in the rest of this subsection.

RQ1 *Can we identify API migration patterns from public code examples?*

Our approach uses code examples to automatically learn API migration patterns. Therefore, the availability of migration code examples is very important to our approach. In this RQ, we study how many API migration patterns we can find in publicly available code examples and calculate the *precision* of our approach. We also engaged the help of 20 volunteer developers to produce a sample migration dataset in order to obtain a known number of migration examples and determine the *recall* of our approach.

RQ2 *To what extent can our approach provide assistance when migrating APIs?*

In this RQ, we use the publicly available and user produced examples to automatically learn and apply migration patterns. We evaluate and quantify the extent to which our approach, A3, can leverage these examples to assist developers in migrating API calls.

RQ3 *RQ3: How much time can our approach save when migrating APIs?*

In this RQ, we enlisted 15 developers to participate in a user study to evaluate the time savings provided by A3. We timed the participants while they migrated an amalgamation of six API examples, for which they randomly received help from A3. The participants were also asked to rate the usefulness of our approach.

4.2 Data acquisition

In this subsection, we present our data acquisition approaches for the Android apps that we want to migrate and the sources of our code examples.

4.2.1 Android apps for migration

We selected our Android apps through the free and open source repository of Android apps: FDroid [16]. We clone all the git repositories of FDroid apps that are hosted on GitHub and implemented in Java. In total, we obtained 1,860 apps. From these apps, we would like to perform API migration on the ones that are still actively under development, since they are more likely to benefit from migrating to the most recent Android APIs. Therefore, we only selected apps that had code committed in the six months prior to our study. In order to assist in later verifying the success of the migration, we selected only the apps that contained readily available tests. Furthermore, we only selected the apps that could be built with the official Android build system, Gradle, so that we could verify the functionality of apps after migration. We focused the evaluation of our approach for API migration on the 10 latest versions of the Android API since they account for 95.6% of Android devices in the world [23]. Therefore, we selected the apps that target the 10 latest versions (API versions 19-28) of the Android API. This allowed us to collect a sample of 164 FDroid apps on which to test our migrations. All the other apps (1,696) were used to extract API migration examples.

4.2.2 Sources of public code examples

Our approach relies on code examples to learn API migration patterns. In the evaluation of our approach, we focus on two sources of public code examples: 1) official Android examples, i.e., Google Samples [12] and 2) FDroid app development history. We also use Android API usage patterns extracted from our FDroid sample to construct the original examples given to our user study participants.

Google Samples. The Android development team provides code samples to assist app developers understand various use cases of the Android APIs. This code sample repository, aptly named Google Samples, contains 234 sample projects, 181 of which are classified as Java projects [12]. We mined the Google Samples repository, hosted on GitHub, for Android API migration examples.

FDroid app development history. Due to the widely available open source projects, if one open source project migrates a deprecated API in their source code, other developers may leverage that migration as an example. With the publicly available development history of FDroid, we can leverage the API migrations that exists in the FDroid apps as code examples.
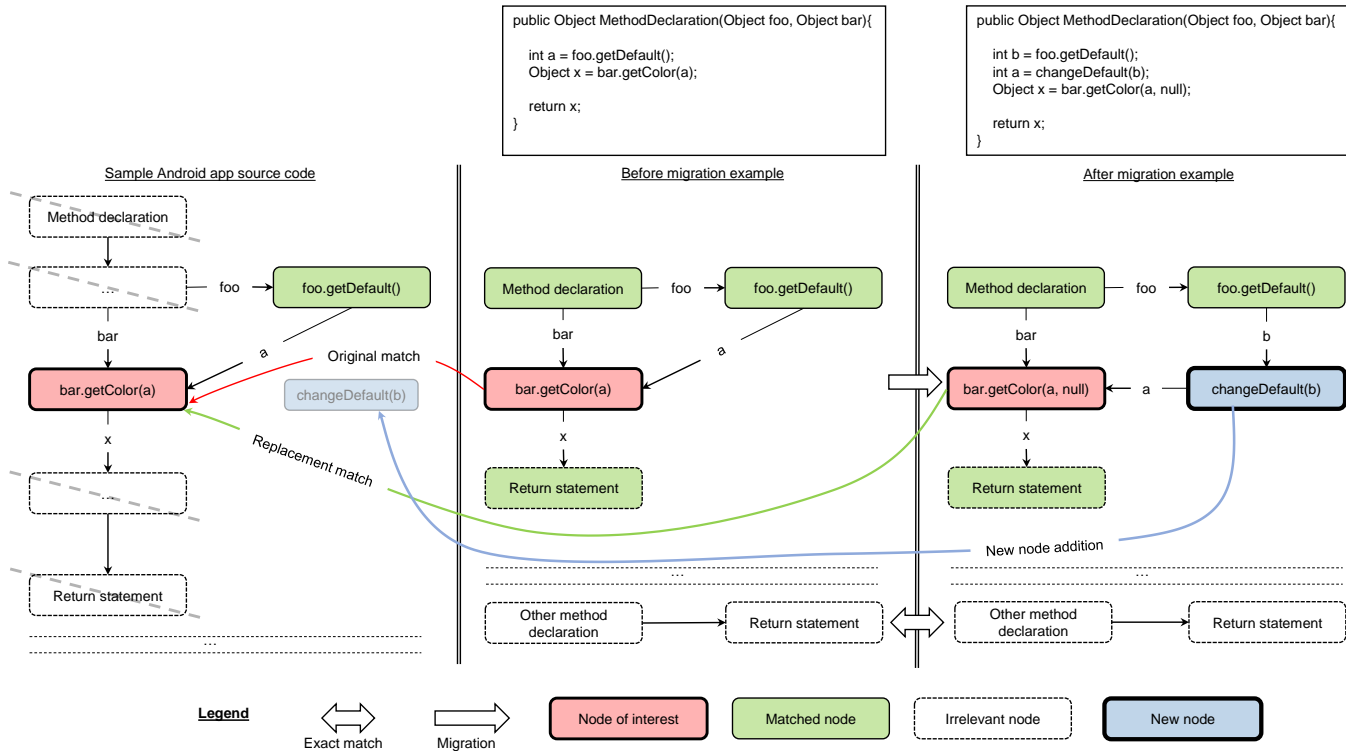
Fig. 3. Example of applying a migration from an example to Android app source code

4.2.3 Source of private code examples

Since it is impossible to determine how many migrations have been produced and can therefore be recovered from mining public repositories, we produced a sample migration dataset with the help of volunteer developers. This dataset was used in RQ1 to determine the recall of our approach when mining for migration examples.

We developed a dataset with the help of 20 participants (14 graduate students and six professional developers) to create migration examples for Android API methods that had known migrations. Based on our study of the FDroid repositories, we selected the 30 Android APIs which were most frequently called but did not have readily available public examples of migration in FDroid apps. We manually created stub classes based on multiple examples of existing code taken from FDroid apps to reduce the amount of code our volunteers had to read. We then presented these classes to our participants as files sampled from real apps that needed migration. The participants were each given five files, selected at random from our pool of 30 Android APIs that needed migration, in order to guarantee multiple samples of successful migrations for each API. Therefore, each API had at least three participants attempt the migration. The participants were told which API to migrate for each file, and were allowed to use the official documentation or any other source of information they deemed necessary to create suitable migrations. We manually verified the results of the participants and randomly selected working examples to seed a mine-able sample repository of 30 real migration code examples that were created by real develop-

ers and sampled from real projects [3].

5 EVALUATION RESULTS

In this section, we present the results of our evaluation by answering three research questions.

RQ1: Can we identify API migration patterns from public code examples?

We focus on two sources of examples in this RQ: 1) official Android examples, i.e., Google Samples [12] and 2) FDroid app development history. We measure the prevalence of API migration examples by determining the number of API migration examples that we can mine from our example sources. We only consider the APIs with migrations that have officially been documented by Android developers to validate our results.

In the 10 latest versions of Android (versions 19-28), we were able to manually identify 262 APIs which had documented migrations. Out of all the Android APIs with migrations, only 125 of those occurred in our sources of examples.

By searching for API migrations (c.f. Step 1 in our approach in Section 3.1) in the Google Samples [12] and the development history of FDroid apps, our approach can automatically identify 10 and 79 API migration examples out of 125 API occurrences, respectively. Among those migration examples from Google Samples, only one API is not covered in the FDroid examples, giving us a total of 80 distinct API migrations found (see Table 1). However, we note that the nine other examples from Google Samples that

3. The 30 code examples, and their migrations, are available as part of our replication package

TABLE 1
API migrations identified

Data sources	API identified
FDroid API migrations	79
Google Samples API migrations	10
Total distinct API migrations found	80
Total API uses found in apps	125
Total possible	262

overlap with other sources are still useful due to the possible minor variations in ASTs that can occur when migrating. Some examples may provide a more suitable migration with less needed human effort from developers (c.f. RQ2 & RQ3). We manually analyzed all examples mined and determined that our approach was able to find migration patterns with 100% precision in Google Samples (i.e., 10/10 examples correctly identified), and 96.3% precision in FDroid apps (i.e., 79/82 examples correctly identified), giving a total precision of 96.7%. Some examples are migration false positives due to user modifications that can be mistaken an migrations, we discuss examples that are not migrations in detail in RQ2.

To calculate the recall of our approach, we used the manually created sample of 30 Android migration examples that was created by volunteer developers (c.f. Section 4.2.3). Our approach was able to successfully extract 29 out of 30 migration examples from the seeded repository, giving us a recall of 97%. The example which could not be extracted was due to line-breaks within the API call. It would be possible to fix this corner case either by preprocessing all commits to remove line-breaks, or modify our API search to allow for arbitrary line-breaks.

TABLE 2
Classified API migration patterns identified by our approach

Migration difficulty	Type	Frequency
EASY	Encapsulate	6
	Move method	4
	Remove parameter	1
	Rename	7
MEDIUM	Consolidate	7
	Expose implementation	10
HARD	Add contextual data	23
	Change type	15
	Replaced by external API	7

We manually classified the 80 distinct migration patterns identified by our approach according to pre-established API migration classifications [22]. Table 2 presents our classified patterns. As can be seen in Table 2, the majority (45/80) of the migrations we identified are considered hard to automate [22]. This implies that our approach can surmount hard technical challenges, as well as find easy-to-migrate examples. The easy difficulty migration implies that the migration patterns can normally be completed by most IDEs such as Eclipse [22]. For example, in API version 26, *Notification.Builder.setPriority* was moved and renamed to *NotificationChannel.setImportance* and *CookieSyncManager.startSync* is no longer useful since it was encapsulated by the WebView class in API version 21 and is now automatically done by that class. Medium difficulty migrations (i.e., migrations previously considered partially automatable) were also found. We identified examples of API consolidation such as *NetworkInfo.getTypeName* which was consolidated

into the *NetworkCapabilities* class in API version 28 and must now instantiate that class and call a different method *hasTransport* new with constants. We also identified cases where implementation was exposed to give more control to the API users. For example the *Notification.Builder.setDefaults* method started allowing users to enable vibration, enable lights and set sound separately in API version 26 rather than be done as part of one method, as it was done before API version 26. This means that our approach must be able to find code to instantiate all of these new functionalities. As previously mentioned, our approach also identifies hard to automate migration patterns. We identify patterns such as added contextual data in methods like *Hmtl.toHtml* where new parameters were added in API version 24 to allow for more options and user control. In most of these cases a default value for new parameters is allowed, which our approach can mine from existing projects to allow users to migrate their project. Changed types such as faced when migrating to *Message.getSenderPerson* from *Message.getSender*, in API version 28, can be handled by our approach by following the control flow graph of calls and modifying the calls appropriately based on previous migrations. Finally, APIs replaced by external APIs such as the FloatMath APIs in API version 22 *cos, sin, sqrt* can be changed to their new library format, and the import statement can be created. The user then only has to make sure that the new API library is included in their project.

> *Our approach can automatically identify 80 migration patterns with 96.7% precision in Android APIs used in public code examples, and obtains a recall of 97% using our seeded repository.*

RQ2: To what extent can our approach provide assistance when migrating APIs?

In this RQ, we examine whether our approach can provide assistance to API migration in Android apps. In particular, we apply our approach to migrate 80 API calls in 32 FDroid apps (based on both the migration patterns learned from public code examples and user generated examples). In order to examine whether a migration is successful, we leverage the tests that are already available in the apps and collect the ones that can exercise the migrated API calls. In order to avoid tests that are already failing before the migration, we run all the tests before the migration of the apps and only keep apps with passing builds. The apps that we selected had an average of 10 tests, with a minimum of 6 tests and a maximum of 52. Afterwards, we try to build the migrated apps. For the apps that can be successfully built, we run the collected tests again to check whether the migration is completed successfully. Furthermore, we also manually install and run the migrated apps to ascertain that the migration does not cause the app to crash. Furthermore, we provide comparisons with prior approaches such as LASE.

Previous studies have shown that producing exact automated migrations is a difficult task [22]. We consider this fact and attempt to mitigate it through the use of our approach. However, in cases where it is possible to present an exact migration to a user, such a task should be attempted to save development time. Therefore, in the best case scenario

we attempt to provide an exact and automatic migration to users. Table 3 summarizes the results of our migrations. Our approach can provide assistance in 71 out of 80 migrations through faultless migrations, migrations with minor modifications, and through the examples we suggest when we experience unmatched guidance or unsupported cases. The following paragraphs discuss each different type of result in detail.

TABLE 3
Automated migration results based on migration patterns learned from three different sources

	FDroid	Google Samples	Volunteer developer sample
Faultless migration	5	0	9
Migrated with minor mod.	5	4	12
Unmatched guidance	18	1	9
False positive	0	0	3
Ex. was not a migration	6	0	0
Unsupported cases	7	0	1
Total migrations	*41*	*5*	*34*
Distinct API	*21*	*4*	*15*

Faultless migration. If all the tests are passed, and the app runs, without any further code modification, we consider the migration as exact. We succeeded at giving users an exact migration in 14 cases. Nine of these migrations were learned from the manually produced code examples from the examples manually created by participants. However, we were able to use five examples from FDroid app development history to produce faultless migrations. Such a result tells us that given a rich example, it is possible to provide fully automated working migrations.

Migration with minor modifications. In cases where an app, after the migration, does not build and run, or does not pass the tests, we manually checked the error message. In such cases, the migration pattern may be correct, yet our automatically generated migration may need minor modifications to build and run the app and pass the tests. For each case, we determined a way for the migration to succeed with minimal code modification. An example of such modification can be adjusting a variable name. We were able to provide 21 migrations with minimal modifications. We consider the number of tokens that must be changed for the migration to be successful. The number of tokens changed was determined by post-modifications performed by the first author, who manually went through failed automated migration cases, modified the app's code to make the migration successful and measured the size of the modification. We use the absolute number of tokens changed rather than a percentage of tokens matched since the automatically matched tokens do not require any effort. We consider any modifications to a code token as a token modification.

We found that the modifications needed for an imperfect but successful migration requires modifying between one to seven tokens (3.65 tokens on average). We consider the amount of effort needed on such modifications rather small, especially since they are mostly simple renames and the addition or removal of keywords. The brunt of the work, namely finding a migration candidate, finding a migration pattern, and matching this pattern, is provided by our approach, A3. Therefore, the user can simply "glue" any unattached pieces of example code into their application.

We examined the different scenarios that require minor modification to qualitatively understand the effort needed for such modifications. In total, we identify four reasons for such modifications: *variable renaming*, *missing the keyword **this***, *wrongly erasing casting*, and *removing API calls*.

As a design choice, we opt to conservatively not remove any API calls for migration, since mistakenly removing API calls may cause large negative impact to developers. Instead, our approach tells the users that a change must be made to the API, and the API call must then be manually removed. Although this may require the manual modification of removing several tokens, the effort of the change is minimal. We experienced three *removing API call* cases. For example the *View.setDrawingCacheQuality(int)* method was made obsolete in API 28 due to hardware-accelerated rendering [24]. This means that old code referring to drawing cache quality can be simply deleted, as the OS now handles this through hardware.

Unmatched guidance. It is possible for our approach to fail to match an example to a known migration (see Section 3.2-1). These cases exist due to the nature of our example matching. Since we consider the data-flow surrounding a migration, our examples must contain a similar data-flow graph to the Android app considered for migration. We conservatively opt for such an approach to reduce the number of falsely generated migrations, since they may introduce more harm to developers than assistance. As a result, if the Android app instantiates their API call in a different way than our examples, our approach will not attempt to automate the migration. For example, if the user uses nested method calls inside an API call and we cannot reliably map the return types of the nested method calls, we will not consider the code to match. Examples that contain unmatched but similar API migrations will be presented to the user, and they can choose the correct migration and manually apply it afterwards.

For the 28 migrations for which none of our examples contained a match, 18 of these are from FDroid development history. This implies that the FDroid API usage is more often tailored for the apps' needs and is not often coded in a general manner. Future research may investigate automatically generalizing the usage of API to address this issue.

False positives. We consider a migration to be a false positive when our approach presents an unnecessary migration. This would occur if a migration example were erroneously matched to a non-migrateable method invocation in Android app code. This only occurred three times in our tests and the occurrences were all from the manually produced examples. We believe this to be the case due to the simplicity of the manually produced examples. Since the examples are simplified and contain very little context code, the corresponding data-flow graphs are often simple. This allows the examples to be used in a wider range of situations than the mined code samples. However, this leads to false positives as a trade-off, especially if method names are commonly used and few parameters are used (e.g. *setContent()*). These false positives can be caught at compile time since they would not compile, and can therefore be corrected or discarded by the developer without harm to

the app.

Example was not a migration. As previously mentioned, since we obtain our migration examples through automated tool assistance (c.f. RQ1), it is possible for our approach to present faulty migration examples. This normally occurs when a deprecated API call is modified, but not migrated to the updated version of the API. An example of such a modification would be to rename the variables in the API call. This change would be mistakenly identified by our approach as a modification to an API call. Therefore, if such an example were to be used to attempt a migration, we would present a migration with no effect, and therefore cause no harm to the Android app. We experienced six such instances in the FDroid examples. However, we did not encounter these in our Google Samples nor the manually produced examples.

Unsupported. There exist a few unsupported corner cases that our approach would not support. For example, we cannot automatically produce migrations if the API call is spread across a try catch block, or within a loop declaration or conditional statement declaration. As with unmatched guidance cases, we provide the user with migration examples that may be relevant to their migration, instead of attempting to provide automated migration on their code, therefore no harm is done to the code. We experienced eight such cases.

TABLE 4
Comparison with LASE [17]

	A3	LASE
Total Migrations Possible	17	
Migration map successfully created	17	15
Migration point successfully found	9	1
Migration faultlessly applied	7	1

Comparison with other approaches. To the best of our knowledge, there currently exists no other approach that can automatically identify API migration examples and automatically suggest and apply them to app code. Tools such as SEMDIFF [9], and AURA [8] find migration locations, but do not provide example-based automatic migrations. We find 97% of migration locations based on our examples, which is close to the upper bound of what these approaches can achieve. We selected LASE [17] as a more direct comparison for our approach since LASE resembles the automatic migration of our approach while needing to be provided compilable examples. Due to LASE's need for compilable examples, we had to use A3 to first obtain migration examples to feed into LASE. Furthermore, LASE does not automatically determine a match between examples and must be manually pointed to paired examples through manually edited configuration files, which we had to produce for each example set. Other similar approaches are discussed further in Section 6. Due to the high degree of manual effort when manually producing configuration files for LASE, we chose a subset of 10 of the 32 apps used for RQ2 to compare with LASE. The apps were selected at random, and contained a total of 17 migration possibilities comprised of 13 distinct APIs. The results can be seen in Table 4.

LASE can build a refactoring map for 15 (88%) migra-
tions compared to 17/17 for A3. However, only 1/17 migrations were automatically mapped to migration points in the apps and subsequently applied to the apps. LASE appears to be highly dependent on the quality of the example and their similarity to the app code when compared to A3. We also find that LASE highly depends on complete matches in method AST. For example, if app code is inside an *if* statement or *try* block when the example code is not, if app code is instantiated differently than example code, or part of a method argument chain is in a different order to the example code, the AST sub-trees of the code will appear to be different. The differences in the sub-trees may lead to mis-detections between the primary example and the app code by LASE. We believe that since LASE was not designed for our use cases, it is intuitive that LASE is not optimal for our API migration task. On the other hand, A3 is designed to mine examples from the online code example, and hence is less sensitive to factors such as as complete AST-alignment and therefore can more often provide migration links when examples are loosely aligned.

We conducted four semi-structured interviews concerning three open source apps from FDroid (*Antennapod, K9Mail, and Wikipedia*). We contacted app developers that were contributors to the apps. Four developers offered to be interviewed for our study. Two developers for the Wikipedia app, one for Antennapod, and one for K9Mail. All of the developers had several years of experience in programming and Android app development. We presented each developer with migration candidates from the app they were familiar with. We also allowed them to search the web, and ask any clarifying questions if they had any. We then provided one example of a migration of the same API call done in another app, mined by our approach. We asked four developers the following questions:

1) "Do you think our approach correctly identified the migration candidate?"
2) "How confident would you be when migrating this API?"
3) "How do you feel about the examples provided by our approach to help you migrate?"
4) "Do you think there are limitations or potential improvements to our approach?"

In all cases, the developers said that we had indeed found useful migration candidates. The migrations were judged to be easy, but time consuming. In all cases the developers said that our examples were *"extremely useful"* or *"very very useful"*, and that they would reduce the time to complete a migration from *"a few minutes, down to a few seconds"*.

One developer said that *"[...] where a new parameter is introduced it's hard to tell if the default value will do the trick, so I would likely have to do a decent amount of research before I felt secure in my choice [...]"*.

One developer identified a potential improvement when providing before/after examples rather than a complete migration, *"When you provide before/after files for a migration it would be nice to have a quick summary of the migration, like did you add a parameter? Otherwise I have to look at the documentation to make sure I'm looking at the right thing, and takes a little bit of extra time."*

TABLE 5
Results of A3 user-study: comparing the time needed to migrate Android API usage examples (measured in seconds) with help from A3 and location-based API migration tools [8], [9].

Example	Avg. time w/o. A3	Avg. time w. A3	Time Improvement (%)	Avg. example usefulness
1	304.9	266.8	12.5	4.2
2	150.0	115.8	22.8	4.5
3	265.5	174.6	34.2	4.8
4	572.8	372.6	34.9	3.9
5	152.4	129.9	14.8	4.8
6	179.8	81.4	54.7	3.8
Total:	270.9	190.2	29.0	4.3

In one case a developer said that, *"Small examples are nice, but sometimes you want more context, so maybe having the whole file is better"*, on the other hand another developer said that,*"I prefer when I can only see the migration and a little surrounding code, if there's too much code, it's harder to see exactly what's going on."*. From these interviews we believe that there is an element of developer preference when looking at examples, and perhaps future research can look at the kinds of examples developers like, and create tools that can determine how much of data-flow and control flow is shown based on developer preference.

> *A3 can provide API migration assistance in 71/80 cases. We can automatically generate 14 faultless migrations, 21 migrations with minor code changes, and 36 migrations with useful guidance to developers. The effort needed to post-modify our generated API migration is low, an average of 3.65 tokens require modification.*

RQ3: How much time can our approach save when migrating APIs?

In this RQ, we present the design and results of a user study involving the assistance provided by our approach. We conduct the user study to evaluate the usefulness of migration suggestions provided by our approach. The user study involves 15 participants (6 professional developers and 9 graduate students). In particular, we compare the time used for API migration by using our approach, and by only using location-based API migration tools, such as SEMD-IFF [9], and AURA [8]. We do not compare to LASE [17] in this section since LASE cannot identify migration cites or examples without developer input. Therefore, comparing A3, which automatically identifies a migration location, and automatically identifies potential migration examples, to LASE which can only migrate manually identified migration locations with manually identified examples, would not be a fair comparison of time saved. Rather, we seek to determine how much time can be saved by an approach that can automatically find migration examples and migration candidates when compared to approaches that only identify migration candidates.

We extracted six API migration tasks from the out-of-date Android API uses from FDroid projects. Each participant was given three tasks with the help of migration suggestions provided by A3, and three other examples with the help of location-based API migration tools, such as SEMDIFF [9], and AURA [8]. We randomized the order of the tasks for each participant. These tasks were part of the dataset used in RQ1 and RQ2 and had been detected by A3 as APIs which needed migration. The tasks were chosen randomly from our sample to avoid bias and we manually ensured that each task did need migration. We used the approach presented in Section 3.2 to obtain a migration suggestion for each of the six tasks.

Each participant was given six source files that presented code with old versions of the Android API. The participants were told that they should attempt to modify the source code in each task to migrate to the latest version of the Android API. For the tasks that receive the help by A3, we provide the code that is generated by A3 after migration. For the tasks that only received the help from location-based API migration tools, the participants were informed of *which* API call to migrate and *what* is the new version of the API to migrate. We also provide the hyper-links to the Android developer website pages necessary to understand the API calls and their migrations were also given to the participants. We note that we did not directly run SEMD-IFF [9] or AURA [8] to obtain above information, but directly provide the ground truth information to the participants, as if SEMDIFF [9] or AURA [8] generate perfect results. Furthermore, the participants were told that we provided potential solutions in the form of migration suggestions for some examples, and that they should attempt to use them if they could. This was done to minimize the noise in the measured time from other activities to concentrate on the migration of API calls themselves.

The participants were timed to determine how long each migration took. They were also asked to rank the usefulness of the migration examples whenever possible. The rank is on a scale of 1 to 5, where 1 was considered as useless, and 5 was considered extremely useful. The results of our user study are presented in Table 5.

Overall, our approach provides an average time improvement of 29% with a p-value of 0.015 in a two-tailed Mann-Whitney U test. Professional developers improved by 45.9% while graduate students by 22.7%. We have therefore shown that automatically providing migration examples to users using a technique like A3 can improve migration times. As shown in Table 5, the developers involved with our user-study also found the assistance from A3 useful, ranking it an average of 4.3 out of 5 in usefulness. Therefore, our approach not only provides migration aid that can reduce migration time, but the examples provided are also judged as useful by developers.

> *Our approach provides, on average, a 29% improvement in API migration speed compared with location-based API migration tools. Users ranked the help provided by A3 an average of 4.3 out of 5 on a usefulness scale.*

6 RELATED WORK

In this section we discuss prior research in the field of APIs. We concentrate on prior work based on API evolution, API migration, and the usefulness of code examples for APIs. Table 6 shows the major differences between our approach and prior studies.

API evolution studies

As the world of software APIs expanded so did the number of studies on API usage and evolution [2], [30]–[45].

TABLE 6
Main differences and novelty provided by the A3 approach when compared to related work.

Related Work	Novelty provided by A3
CatchUp! [25]	Our approach does not require recording of API modifications.
SemDiff & AURA [26], [27]	Our approach uses API usage examples rather than internal API modifications to obtain migration patterns.
EXAMPLORE [28]	Our approach automatically obtains and applies relevant example code as patterns for migration.
Sydit & LASE [17], [18]	Our approach automatically obtains matched code examples, does not require fully compilable projects.
LibSync [29]	Our approach gives fully migrated code, and does not require fully compilable projects.

More recently, several research papers have been published on the Android API and its evolution [5], [6], [46]–[53]. The Android API is both large in code size and has numerous versions. It has therefore been found to be a suitable case study for various types of software research, from API usage [54] and crashes [55], to software testing [56], [57], Stack Overflow discussions [58], static analysis tool creation [59], and user ratings [4]. We use the Android API as a case study for API evolution and migration.

In 2013, McDonnell et al. [5] extracted the change information of the Android API. They found that the Android API is rapidly evolving and provides an average of 115 API updates per month. API updates are said to occur to fix bugs, enhance performance, and to respect new standards [5]. They suggest that although there are currently many tools to automate the API updating process, these tools are insufficient for current needs and that new studies and tools are necessary to encourage proper API updates [5]. The work by McDonnel et al. is a strong motivation for the ideas presented in this paper. Due to the rapid nature of changes in the Android API, it is necessary to have approaches such as the one presented in this paper to aid developers in adapting API changes.

Li et al. [46], [53] built a tool to investigate deprecated Android APIs. Li et al. find that 37.87% of apps make use of deprecated APIs [46]. This knowledge further strengthens the need for tools to help developers migrate away from these methods. They also found that although the Android framework developers consistently document replacements for deprecated APIs, this documentation is not always up to date [46]. This presents a perfect opportunity for our approach to use the most up-to-date to help users migrate away from deprecated API rather than rely on potentially outdated documentation.

API migration techniques

Past research has presented tools and suggestions to improve API migrations [8]–[10], [60], [60]–[67]. API migration tools usually rely on fully automatic or semi-automatic means to provide API migration recommendations to developers. These approaches concentrate on locating a migration candidate and recommending an alternative API. Meanwhile, our approach assumes that the user knows that an alternative API exists, either through these tools or through documentation, and does not understand *how* to produce the migration. This is where our approach, A3, aims to guide users.

CatchUp! [10] provides a semi-automatic migration solution which requires that API developers record API source code changes from their IDE. These changes can then be replayed on API user systems to adapt their application to the changed API. This tool can provide safe migrations due to the nature of the tool chain. However, the tool requires the API developers to record their own work. Contrarily to *CatchUp!*, A3 uses a myriad of examples to guide the migration from any available source.

Dagenais and Robillard, and Wu et al. provide different semi-automatic migration tools, *SemDiff* [9], and *AURA* [8]. Both tools similarly mine method changes in two versions of an API, and internal adaptations to these API changes. These changes are then used to provide migration recommendations to API users. *SemDiff* presents migration scenarios by heuristically ranking the most likely candidates. Meanwhile, *AURA* uses heuristics to provide only the best fit migration. A3 can consider any source of example to extract migrations. This allows users to obtain high quality migration examples even if the API source code is proprietary, as long as open source examples of migrations exist or can be created.

Similarly to A3, more recent approaches such as Meditor [68] and the work by Fazzini et al. [69], attempt to mine repositories to obtain examples and apply those examples to real world applications. Meditor [68] concentrates on Java applications and manages to correctly apply code migrations in 96.9% of cases. Other similar approaches [69] have since been attempted on the Android ecosystem with a success rate of 85%. However, to the best of our knowledge, our approach is the only one so far to have included evaluation in real world scenarios with a user study and developer interviews.

Code examples and APIs

A3 relies on the existence of source code examples of API migrations. Prior research has covered API documentation enhancement and API examples extensively [70]–[83].

Gao et al. find that crypto-APIs misuses are very common in Android apps [84]. They find that developers are unlikely to successfully fix these misuses when they attempt to do so [84]. These findings highlight the importance of proper testing and keeping the developer in the development loop. This is one of the reasons why A3 is not a fully automated approach, but rather allows the developer to have the final say in any migration decision. Therefore, it is always possible for the developer to determine the validity of any migration before choosing to use it.

EXAMPLORE is a visualization tool to assist users in understanding common uses of APIs [28]. The tool uses a corpus of API examples to build common usage patterns which can then be displayed to a user in an interactive graphical way. Since tools like *EXAMPLORE* have shown that they can assist users to answer API questions with detail and confidence [28], we believe that using examples to provide API insights could similarly be used for a migration tool.

Tools such as *Sydit*, *LibSync*, and *LASE* provide systematic editing through examples [18], [29], [74]. However, these tools require well-built ASTs for dependency analysis, but examples of API migration may not always be compilable (e.g., code snippets from documentation, or examples

from users). Furthermore, a developer has to provide the examples with relevant context, or in the case of *LibSync* project versions where migrations are known to occur (e.g., where to apply the changes). In contrast, our approach automatically mines relevant examples, and then finds the most relevant migration pattern for a specific API migration use case with no dependency assumptions (i.e., the code can be non-compilable).

MAPO is a tool that allows developers to rapidly search for frequent API usages [85]. MAPO uses a combination lightweight source code analysis, code search engines, and frequent item-set mining methods [85]. Tools like *MAPO* have shown that it is possible to rapidly and reliably find examples of API usage on the web. Therefore, tools like A3 can rely on this foundation to use these examples for more complex tasks, such as aiding API migration.

We do not provide direct comparisons with other approaches as the work presented in this paper does not directly compare to previous approaches but rather augments them. It would be possible to use the example gathering power provided by A3 and the example matching approaches provided by Sydit, LASE, or LibSync to complete the migration process with minor modifications to these approaches. However, in our partial and indirect comparison with LASE, SEMDIFF [9], and AURA [8] we find that our approach matches or outperforms prior approaches. The primary novelty of our approach is getting migration patterns directly from publicly available sources without needing the compilation of the sources, or developer intervention, and directly using these sources to determine if a migration pattern can be used to modify developer code, and then automatically attempting the migration and providing this to the user. No other approach to date covers the full scope of our approach and therefore no direct and fair comparison can be made.

7 THREAT TO VALIDITY

Construct validity. We assess the validity of our API migrations by building and running the apps as well as running the test suites of the migrated Android apps. Although we focus on the tests that exercise the migrated API calls, and attempt to exercise as much functionality as possible when running the apps, it is still possible that defects introduced by the migration are not identified by our tests. User studies and interviews with developers may complement the evaluation of our approach. In our study, there are still cases where our approach cannot migrate faultlessly. Although our approach can provide migration guidelines to developers, as an early attempt of this line of research, our approach can be further complemented by other techniques such as code completion to achieve better assistance in API migration. Furthermore, we concentrate on the majority of cases through file level migrations. If a large number of migrations occur across multiple files, our approach is not currently able to help.

External validity. Since this entire study was tested on the Java API of the Android ecosystem, it is possible that the findings in this paper will not generalize to other programming languages. However, while it is true that the approach presented in this paper was tested specifically on a Java

based API, all of the approaches are built upon assumptions that are true in other popular programming languages such as C#.

Internal validity. Our findings are based on the Android project and code examples mined and produced for its API. It is possible that we only found a subset of all migrations. It is also possible for the time gap between the release of new API and the update to examples to be larger in other sources. We attempted to mitigate these threats through mining official samples, open source projects, and having participants produce examples for frequently used APIs. We found that Google Samples updated deprecated API as soon as one month after the release of a new API version, which should allow developers to regularly update their apps. Our participant created examples were new and useful API migration examples, showing that the premise of using examples to help automate API migrations is functional and likely dependent on the sample size of examples.

8 CONCLUSION

In this paper, we proposed an approach that assists developers with Android API migrations by learning API migration patterns from code examples mined directly from available code repositories. We evaluate our approach by applying automated API migrations to 32 open-source Android apps from FDroid and through a user-study. We find that our approach can automatically extract API migration patterns from both public code example and manually produced API examples that are created with minimal effort. By learning API migration patterns from these examples, our approach can provide either automatically generated API migrations or provide useful information to guide the migrations. Our user-study showed that the examples provided by our approach allow users to migrate Android APIs, on average, 29% faster and are seen as useful by developers, who ranked them an average usefulness of 4.3 out of 5.

This paper makes the following contributions:

- We propose a novel approach that learns API migration patterns from code examples taken directly from available code repositories.
- Our novel approach can automatically assist in API migration based on the learned API migration patterns.
- We produce a user study and conduct semi-structured interviews that conclusively shows that migration examples are both desirable and useful to developers.

Our approach illustrates the rich and valuable information in code examples that can be leveraged in API related software engineering tasks.

REFERENCES

[1] R. L. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE Software*, vol. 18, no. 3, pp. 112–111, 2001.
[2] P. K. Venkatesh, S. Wang, F. Zhang, Y. Zou, and A. E. Hassan, "What do client developers concern when using web apis? an empirical study on developer forums and stack overflow," *2016 IEEE International Conference on Web Services (ICWS)*, 2016.

[3] A. A. Sawant and A. Bacchelli, "fine-grape: fine-grained api usage extractor – an approach and dataset to investigate api usage," *Empirical Software Engineering*, vol. 22, no. 3, p. 1348–1371, Apr 2016.

[4] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of API change- and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2015.

[5] T. Mcdonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," *2013 IEEE International Conference on Software Maintenance*, 2013.

[6] M. Lamothe and W. Shang, "Exploring the use of automated api migrating techniques in practice: An experience report on android," *MSR '18: 15th International Conference on Mining Software Repositories*, 2018.

[7] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK," *Empirical Software Engineering*, pp. 1–40, 2017.

[8] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: A hybrid approach to identify framework evolution," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE 10*, 2010.

[9] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," *2009 IEEE 31st International Conference on Software Engineering*, 2009.

[10] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005.

[11] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, 2016.

[12] "Samples, android developers." [Online]. Available: https://developer.android.com/samples/

[13] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," *Proceedings of the 38th International Conference on Software Engineering - ICSE 16*, 2016.

[14] S. Wang, I. Keivanloo, and Y. Zou, "How Do Developers React to RESTful API Evolution ?" *12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings*, vol. 8831, pp. 245–259, 2014.

[15] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in Android apps: an empirical study," *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pp. 2–11, 2014.

[16] "F-droid," 2017. [Online]. Available: https://f-droid.org/

[17] M. Kim, "LASE: Locating and Applying Systematic Edits by Learning from Examples," *ICSE*, pp. 502–511, 2013.

[18] N. Meng, M. Kim, and K. S. McKinley, "Sydit: Creating and applying a program transformation from an example," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 440–443.

[19] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds., *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, 2014.

[20] "Android developers resources," Jul 2017. [Online]. Available: https://developer.android.com/reference/android/content/res/Resources#getColor(int)

[21] W. Beaton, "Eclipse corner article abstract syntax tree, the eclipse foundation." [Online]. Available: http://www.eclipse.org/

[22] B. E. Cossette and R. J. Walker, "Seeking the ground truth: A retroactive study on the evolution and migration of software libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 55:1–55:11.

[23] "Distribution dashboard android developers." [Online]. Available: https://developer.android.com/about/dashboards/

[24] "Package index," Jul 2017. [Online]. Available: https://developer.android.com/reference/packages

[25] J. Henkel and A. Diwan, "CatchUp! Capturing and Replaying Refactorings to Support API Evolution," *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 0–9, 2005.

[26] B. Dagenais and M. P. Robillard, "SemDiff: Analysis and recommendation support for API evolution," *Proceedings - International Conference on Software Engineering*, pp. 599–602, 2009.

[27] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "AURA:A Hybrid Approach to Identify Framework Evolution," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 1, p. 325, 2010.

[28] G. E., Z. T., H. B., and K. M., "Visualizing api usage examples at scale," *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI 2018*, 2018.

[29] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '10*. ACM Press, 2010.

[30] D. Dig and R. Johnson, "The role of refactorings in API evolution," *IEEE International Conference on Software Maintenance, ICSM*, vol. 2005, pp. 389–400, 2005.

[31] D. Mendez, B. Baudry, and M. Monperrus, "Empirical evidence of large-scale diversity in api usage of object-oriented software," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept 2013, pp. 43–52.

[32] K. Taneja, D. Dig, and T. Xie, "Automated detection of api refactorings in libraries," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 377–380.

[33] J. Businge, A. Serebrenik, and M. van den Brand, "Analyzing the eclipse api usage: Putting the developer in the loop," in *2013 17th European Conference on Software Maintenance and Reengineering*, March 2013, pp. 37–46.

[34] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to restful api evolution?" in *Service-Oriented Computing*, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 245–259.

[35] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service API evolution affect clients?" *Proceedings - IEEE 20th International Conference on Web Services, ICWS 2013*, no. 61121063, pp. 300–307, 2013.

[36] C. Jaspan and J. Aldrich, "Checking framework interactions with relationships," in *ECOOP 2009 – Object-Oriented Programming*, S. Drossopoulou, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 27–51.

[37] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[38] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4f: Taint analysis of framework-based web applications," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 1053–1068.

[39] W. Wu, A. Serveaux, Y. G. Guéhéneuc, and G. Antoniol, "The impact of imperfect change rules on framework API evolution identification: an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1126–1158, 2015.

[40] R. Holmes and R. J. Walker, "Informing eclipse api production and consumption," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '07. New York, NY, USA: ACM, 2007, pp. 70–74.

[41] J. Businge, A. Serebrenik, and M. G. J. van den Brand, "Eclipse api usage: the good and the bad," *Software Quality Journal*, vol. 23, no. 1, pp. 107–141, Mar 2015.

[42] C. Jaspan and J. Aldrich, "Checking framework interactions with relationships," in *ECOOP 2009 – Object-Oriented Programming*, S. Drossopoulou, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 27–51.

[43] M. W. Godfrey, "Practical data exchange for reverse engineering frameworks: Some requirements, some experience, some headaches," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 1, pp. 50–52, Jan. 2001.

[44] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "How a good software practice thwarts collaboration: The multiple roles of apis in software development," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '04/FSE-12. New York, NY, USA: ACM, 2004, pp. 221–230.

[45] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Softw. Engg.*, vol. 16, no. 6, pp. 703–732, Dec. 2011.

[46] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated android apis," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 254–264.

[47] M. Mahmoudi and S. Nadi, "The android update problem: An empirical study," in *Proceedings of the 40th International Conference on Software Engineering*, ser. MSR 2018. New York, NY, USA: ACM, 2018.

[48] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 477–487.

[49] P. Calciati, K. Kuznetsov, X. Bai, and A. Gorla, "What did really change with the new release of the app?" in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 142–152.

[50] H. Cai, Z. Zhang, L. Li, and X. Fu, "A large-scale study of application incompatibilities in android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 216–227.

[51] L. Li, T. F. Bissyandé, Y. L. Traon, and J. Klein, "Accessing inaccessible android apis: An empirical study," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 411–422.

[52] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 153–163.

[53] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Cda: Characterising deprecated android apis," *Empirical Software Engineering*, Nov 2020.

[54] W. Wang and M. W. Godfrey, "Detecting api usage obstacles: A study of ios and android developer questions," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, May 2013, pp. 61–64.

[55] P. Kong, L. Li, J. Gao, T. F. Bissyandé, and J. Klein, "Mining android crash fixes in the absence of issue- and change-tracking systems," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 78–89.

[56] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "Mdroid+: A mutation testing framework for android," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 33–36.

[57] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, March 2019.

[58] M. Linares-Vásquez, G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*, 2014.

[59] L. Li, T. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: a systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 8 2017.

[60] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of api usages," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 47–50.

[61] W. B. Langdon, D. R. White, M. Harman, Y. Jia, and J. Petke, "Api-constrained genetic improvement," in *Search Based Software Engineering*, F. Sarro and K. Deb, Eds. Cham: Springer International Publishing, 2016, pp. 224–230.

[62] J. Li, C. Wang, Y. Xiong, and Z. Hu, "Swin: Towards type-safe java program adaptation between apis," in *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '15. New York, NY, USA: ACM, 2015, pp. 91–102.

[63] C. Wang, J. Jiang, J. Li, Y. Xiong, X. Luo, L. Zhang, and Z. Hu, "Transforming Programs between APIs with Many-to-Many Mappings," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 25:1–25:26.

[64] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, no. i, pp. 511–522, 2016.

[65] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining api usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 457–468.

[66] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mapping api elements for code migration with vector representations," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 756–758.

[67] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343.

[68] S. Xu, Z. Dong, and N. Meng, "Meditor: Inference and application of api migration edits," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, May 2019, pp. 335–346.

[69] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 204–215.

[70] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, "T2api: Synthesizing api code usage templates from english texts with statistical translation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 1013–1017.

[71] R. P. L. Buse and W. Weimer, "Synthesizing api usage examples," *2012 34th International Conference on Software Engineering (ICSE)*, 2012.

[72] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the api jungle," *SIGPLAN Not.*, vol. 40, no. 6, pp. 48–61, Jun. 2005.

[73] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, May 2013, pp. 319–328.

[74] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 502–511.

[75] H. Niu, I. Keivanloo, and Y. Zou, "Api usage pattern recommendation for software development," *J. Syst. Softw.*, vol. 129, no. C, pp. 127–139, Jul. 2017.

[76] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen, "Statistical learning of api fully qualified names in code snippets of online forums," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 632–642.

[77] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "ExPort: Detecting and visualizing API usages in large source code repositories," *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp. 646–651, 2013.

[78] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 643–652.

[79] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence,*

IJCAI-18. International Joint Conferences on Artificial Intelligence Organization, 7 2018, pp. 2269–2275.

[80] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin, "Automatic model generation from documentation for java api functions," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 380–391.

[81] C. Parnin and C. Treude, "Measuring api documentation on the web," *Proceeding of the 2nd international workshop on Web 2.0 for software engineering - Web2SE 11*, 2011.

[82] H. Zhong and Z. Su, "Detecting api documentation errors," *SIGPLAN Not.*, vol. 48, no. 10, pp. 803–816, Oct. 2013.

[83] W. Wang, H. Malik, and M. W. Godfrey, "Recommending posts concerning api issues in developer q&a sites," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 224–234.

[84] J. Gao, P. Kong, L. Li, T. F. Bissyandé, and J. Klein, "Negative results on mining crypto-api usage rules in android apps," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019, pp. 388–398.

[85] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 54–57.
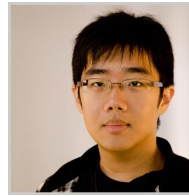
Tse-Hsun (Peter) Chen Tse-Hsun (Peter) Chen is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He leads the Software PErformance, Analysis, and Reliability (SPEAR) Lab, which focuses on conducting research on performance engineering, program analysis, log analysis, production debugging, and mining software repositories. His work has been published in flagship conferences and journals such as ICSE, FSE, TSE, EMSE, and MSR. He serves regularly as a program committee member of international conferences in the field of software engineering, such as ASE, ICSME, SANER, and ICPC, and he is a regular reviewer for software engineering journals such as JSS, EMSE, and TSE. Dr. Chen obtained his BSc from the University of British Columbia, and MSc and PhD from Queen's University. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years. Early tools developed by Dr. Chen were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at: http://petertsehsun.github.io/.

Maxime Lamothe Maxime Lamothe is a Ph.D. student in the department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He has received his M.Eng. degree from Concordia University and he obtained his B.Eng. from McGill University. His research interests include API evolution, API migration, machine learning in software engineering, big data software engineering, and empirical software engineering.

Weiyi Shang Weiyi Shang is an Assistant Professor and Concordia University Research Chair in Ultra-large-scale Systems at the Department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his Ph.D. and M.Sc. degrees from Queen's University (Canada) and he obtained B.Eng. from Harbin Institute of Technology. His research interests include big data software engineering, software engineering for ultra-large–scale systems, software log mining, empirical software engineering, and software performance engineering. His work has been published at premier venues such as ICSE, FSE, ASE, ICSME, MSR and WCRE, as well as in major journals such as TSE, EMSE, JSS, JSEP and SCP. His work has won premium awards, such as SIGSOFT Distinguished paper award at ICSE 2013 and best paper award at WCRE 2011. His industrial experience includes helping improve the quality and performance of ultra-large-scale systems in BlackBerry. Early tools and techniques developed by him are already integrated into products used by millions of users worldwide. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/~shang.