AUSearch: Accurate API Usage Search in GitHub Repositories with Type Resolution

Muhammad Hilmi Asyrofi, Ferdian Thung, David Lo, and Lingxiao Jiang
Singapore Management University
Email: {mhilmia, ferdianthung, davidlo, lxjiang}@smu.edu.sg

***Abstract*—Nowadays, developers use APIs to implement their applications. To know how to use an API, developers may search for code examples that use the API in repositories such as GitHub. Although code search engines have been developed to help developers perform such search, these engines typically only accept a query containing the description of the task that needs to be implemented or the names of the APIs that the developer wants to use without the capability for the developer to specify particular search constraints, such as the class and parameter types that the relevant API should take. These engines are not designed for cases when the specific API and its types to search are known and the developers want code examples that show how the specific API is used, and therefore, their search results are often too inaccurate. In this work, we propose a tool, named AUSearch, to fill this gap. Given an API query that allows type constraints, AUSearch finds code examples in GitHub that contain usages of the specific APIs in the query. AUSearch performs type resolutions to ensure that the API usages found in the returned files are indeed invocations of the APIs specified in the query and highlights the relevant lines of code in the files for easier reference. We show that AUSearch is much more accurate than GitHub Code Search.[1] A video demonstrating our tool is available from https://youtu.be/DKiGal5bSkU.**

***Index Terms*—API usage, code search, type resolution, GitHub**

I. Introduction

Modern software development leverages APIs to build software applications. APIs provide common functionalities to ease software development. Developers may learn how to use an API by reading from the API documentation and tutorials, looking at questions and answers from StackOverflow, or searching for code examples from repositories such as GitHub.

Searching for code examples is among the most frequent activity in software development [11]–[13]. Developers have been found to prefer searching for code examples when they have problems in programming [2], [3]. It allows them to learn directly from pieces of code that should be working and may be able to reuse the relevant code snippets with little to no modification.

To find code examples, one usually uses a code search engine. Code search engines typically accept a textual description of a programming task that needs to be completed as the query and returns code examples in the order of relevance to the query. Popular search engines such as GitHub Code Search simply performs exact keyword matches that return any code containing the names of the APIs specified in the query. These kinds of search engines are not optimized for searching examples of a particular target API with a particular type signature. Performing such search may return many irrelevant code examples that do not actually contain the target API, but rather identifiers whose names match with keywords extracted from the name of the target API, without considering the class and parameter type constraints of the API. Therefore, there is a need for a search engine that can return code examples containing the actual uses of the target API. This kind of search engines requires type resolution. Without resolution, the search engines would be unable to differentiate between APIs with the same method name but different class and/or parameters.

A common use case for the above kind of search engines is when developers have in mind the target API but want to know how it can be used. For example, a post in StackOveflow[2] suggests the API to replace a deprecated API but none of the answers satisfies the questioner's needs on knowing how to use the API. The questioner can instead use our proposed search engine to look for code examples that actually use the API.

The closest existing tool that may be used to find code examples following the above requirement is Boa [4]. Boa is a domain-specific language and infrastructure that eases mining software repositories. Boa supports Abstract Syntaxt Tree (AST) traversal that can be used to find a method with a certain name. However, since Boa does not support type resolution, it is not possible to know the type of the object variable an invoked method belongs to or the types of arguments of the method. As such, it is difficult for Boa to accurately find the code examples that contain the actual uses of the target API.

In this work, we develop a tool named AUSearch to find API usages for a given API query. AUSearch accepts one or multiple API signatures as input and returns the files in GitHub that contain all of the specified API signatures. AUSearch is built on top of GitHub Code Search and overcomes the limitations of GitHub Code Search that only considers textual query and matches any text (even comments) inside a file. AUSearch is also designed to reduce the time required to get the first relevant file from GitHub that contains uses of the specified APIs. Currently, AUSearch only supports Java programming language.

[1] https://github.com/search

[2] https://stackoverflow.com/questions/35647821/android-notification-addaction-deprecated

We have evaluated AUSearch considering the relevancy of its search results and its user-friendliness. We demonstrate that AUSearch can find relevant API usages by filtering out irrelevant GitHub Code Search results. We demonstrate AUSearch's user-friendliness by comparing its result against GitHub Code Search result. We have also released the implementation of AUSearch.[3]

II. API Query Definition

Our goal is to find files that contain uses of all specified APIs in a query and highlight the lines of code surrounding the API uses. To identify such usages accurately, we perform type resolution. Without resolving types, the process of matching API method invocation would be imprecise as the matched API method invocation may correspond to a method with a wrong type and/or use parameters of wrong types.

AUSearch accepts inputs that follow the following grammar:

$$
\begin{aligned}
Query &: mSignatureList \\
mSignatureList &: mSignature\ \{\&\ mSignature\} \\
mSignature &: class\#method(paramList) \\
paramList &: param\ \{,\ param\}
\end{aligned}
$$

In the above grammar, the $Query$ is expressed as a list of method signatures $mSignatureList$. Each signature $mSignature$ in $mSignatureList$ consists of $class$, $method$, and $paramList$. class represents a fully qualified type[4] of a class. $method$ represents the name of the method that belongs to $class$. $paramList$ is a list of parameters for the $method$. Each $param$ in $paramList$ is the fully qualified type of a $method$ parameter.

As a concrete example, developers may want to find code examples that use the `vibrate` method that belongs to `android.os.Vibrator` class and accepts parameters of types `long[]` (first parameter) and `int` (second parameter). In this case, the query is `android.os.Vibrator#vibrate(long[], int)`.

III. AUSearch

A. Architecture

The architecture of AUSearch is shown in Figure 1. AUSearch accepts as input a *User Query* following the API usage query definition provided in Section II and returns *Matched Files*, which are source code files that have been checked to contain invocations of methods that match with the *User Query*. AUSearch consists of four main components: *Query Processor*, *Type Resolver*, *Package Analyzer*, and *Filtering Module*. AUSearch has two main data sources: *GitHub Code Search API* and *Maven Package Search API*.

Query Processor takes *User Query* as input and returns *Java Files* that are obtained from *GitHub Code Search API* as output. *Java Files* are then input to *Package Analyzer* to retrieve *Required Jars* that contain the *Imported Packages* in *Java Files*. *Required Jars* along with *Java Files* and *User Query* are input to *Type Resolver* that resolves the types of variables inside *Java Files* and returns *Resolved Java Files* as the result. *Filtering Module* takes these *Resolved Java Files* and *User Query* as input and returns *Matched Files*.

The details of each main component are as follows.

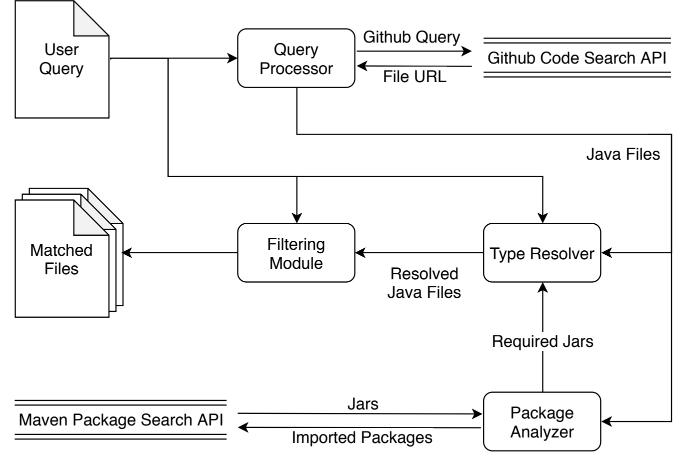

Fig. 1. Architecture of AUSearch

1) Query Processor: Query Processor breaks down *User Query* to tokens that constitutes the *User Query* by removing all symbols including "#", "&", ".", "(", and ")" from the *User Query* to make a *GitHub Query*. It then sends the *Github Query* to *GitHub Code Search API* and receives *File URL*, which is URL of the source code file in GitHub that matches with the *GitHub Query*. *Query Processor* downloads the files and returns them as *Java Files*. *Query Processor* processes *Java Files* one by one and send them to the next components in the pipeline (i.e., *Package Analyzer* and *Type Resolver*).

2) Package Analyzer: Given *Java Files*, *Package Analyzer* parses the file and collects packages that are required by the file. These packages are then input as queries to *Maven Package Search API*. *Maven Package Search API* returns the names of jars (i.e., a collection of Java files that are packaged into a single file) containing the packages. As multiple jars may contain the same package, *Maven Package Search API* returns a ranked list of jars sorted according to *Maven Package Search API* internal algorithm. *Package Analyzer* simply picks the first jar in this order. *Package Analyzer* implements a simple caching mechanism by checking whether the jar exists in the local storage. If the jar exists, *Package Analyzer* does not download the jar as it has been downloaded before. If the jar does not exist, *Package Analyzer* downloads the jar. *Package Analyzer* then returns *Jars* that contains the imported packages inside the Java file.

3) Type Resolver: Type Resolver accepts as input *Java Files*, *User Query*, *Jars* containing the imported packages inside the *Java Files*. *Type Resolver* parses the *Java Files* into an Abstract Syntax Tree (AST). *Type Resolver* then traverses the tree to find invocations of methods in *Java Files* whose fully qualified names are the same with the methods specified

[3] https://github.com/mhilmiasyrofi/search-visualization

[4] In Java, this refers to a package name followed by a class name

in the *User Query*. For each of such method invocations, using information from the *Jars*, *Type Resolver* uses javaparser[5] to resolve the type of the class the method belongs to and types of all method arguments. If all the types can be resolved, *Type Resolver* returns the corresponding files among the set of *Resolved Java Files*. *Resolved Java Files* contain line numbers indicating the API usage locations.

4) Filtering Module: *Filtering Module* accepts as input *Resolved Java Files* and *User Query*. *Filtering Module* checks whether the resolved type of the class the method belongs to exactly matches the type in the *User Query*. It also checks whether the resolved types of the arguments exactly match the types of the parameters in the query. If both checks result in a successful match for each API specified in *User Query* , *Filtering Module* puts the corresponding file in the list of *Matched Files*. Otherwise, it discards the file. *Matched Files* are shown to users with the lines containing the API usages highlighted.

B. User Interface

The user interface of AUSearch is shown in Figure 2. Users type a query insides a text box following the format specified in Section II. AUSearch follows the process described in Section III-A and returns *Matched Files*. In Figure 2, we are asking for `setTextAppearance` method, which belongs to `android.widget.TextView` and has parameters of types `android.content.Context` and `int`. The snippets of matched files are shown below the query text box. A snippet shows a portion of a matched file that contains the queried API usage, in which the API usage is highlighted in yellow. The snippet also contains the file location in GitHub, which is a combination of the repository address containing the file and the file path from the root of the repository. In Figure 2, the repository address is *fireberlin/tinytimetracker* and the file path is *android/src/com/fireberlin/tinytimetracker/ui/LogDailySummaryView.java*. Clicking on the file location would bring users to the GitHub page for the file while clicking on the highlighted line would bring users to the same page with the corresponding line highlighted.

IV. Usage Scenarios

A. Effective Search with Multiple Methods Sharing the Same Name

When searching for Java code examples, AUSearch is especially useful for finding files containing API usages in which the API has the same method name with other APIs that are either from the same class (due to method overloading) or different classes. In such case, developers would find it harder to identify whether a method with the same name in the file is the one they are looking for or not. Without AUSearch, developers would need to manually check whether the type of object and types of arguments are actually match with the API they are looking for.

[5]https://javaparser.org/

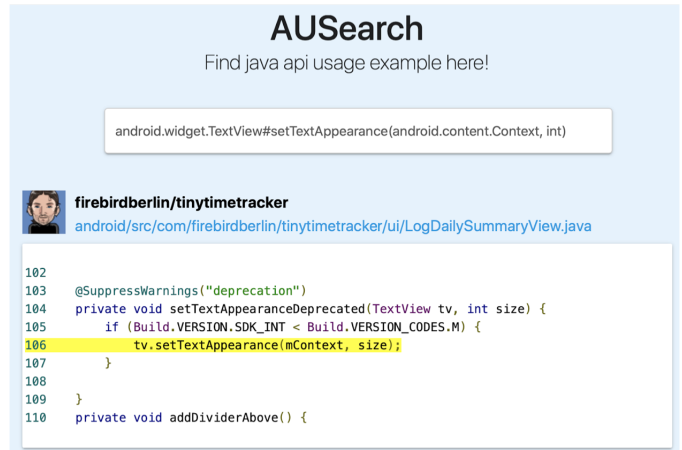

Fig. 2. User Interface of AUSearch

Suppose we look for usages of `addAction` method that belongs to `android.app.Notification.Builder` class with a parameter type `android.app.Notification.Action`. The method is overloaded by another method with three parameter types `int`, `java.lang.CharSequence`, and `android.app.PendingIntent`. AUSearch produces a result as in Figure 3. On the other hand, searching for the same method in GitHub puts the relevant method in the 44th position, which means that developers need to sift through a lot of irrelevant results before finding code snippet containing the API usage. Thus, AUSearch can save much developer time in looking through search results.

Fig. 3. AUSearch Result

B. Accurate Highlighting of Lines containing Method Invocations

AUSearch helps developers investigate returned code snippets faster by highlighting relevant lines of codes, i.e., lines of codes that contain the usages of specified APIs. For this reason, AUSearch is a better choice than GitHub Code Search for user friendliness. GitHub Code Search result would highlight lines of codes that are a match with tokens appearing in a given textual query. For example, suppose that we want to find API usages for `addAction` method, which belongs to `android.app.Notification.Builder` object and has three parameters of types

`int`, `java.lang.CharSequence`, and `android.app.PendingIntent`. GitHub Code Search returns results as shown in Figure 4. Notice that since GitHub Code Search highlights lines of code that contain tokens in the query such as `android`, `app`, and `Notification`. These lines of code may include import statements and class declaration, which clearly do not contain API usage.

AUSearch provides better highlighting capability to ease the task of finding API usages. Figure 5 shows the same result file as in Figure 4 but highlighted by AUSearch. Notice that the lines of code highlighted by AUSearch actually contain the API usage that we are searching for. This is possible since AUSearch performs type resolution and therefore is capable of highlighting the lines of code with actual API usage. Developers' time are saved because they do not need to manually check the file to find the lines containing the API usage, like what they would need to do if they used GitHub Code Search.

Mi-Walkie-Talkie-by-Darkhorse/Mi-Walkie-Talkie-Plus
original-java/android/support/v4/app/NotificationCompatJellybean.java

```
6    import android.app.Notification.BigTextStyle;
7    import android.app.Notification.InboxStyle;
8    import android.app.PendingIntent;
9    import android.content.Context;
-
54       public static class Builder implements NotificationBuilderWithActions,
     NotificationBuilderWithBuilderAccessor {
55           private android.app.Notification.Builder b;
```

Fig. 4. Highlighted File in GitHub Code Search Result

Mi-Walkie-Talkie-by-Darkhorse/Mi-Walkie-Talkie-Plus
original-java/android/support/v4/app/NotificationCompatJellybean.java

```
204         return factory.build(i, charSequence, pendingIntent, bundle, remoteInputArr, z)
205     }
206
207     public static Bundle writeActionAndGetExtras(android.app.Notification.Builder build
208         builder.addAction(action.getIcon(), action.getTitle(), action.getActionIntent()
209         Bundle bundle = new Bundle(action.getExtras());
210         if (action.getRemoteInputs() != null) {
211             bundle.putParcelableArray("android.support.remoteInputs", RemoteInputCompat.
212         }
```

Fig. 5. Highlighted File in AUSearch Result

V. Evaluation

We evaluate AUSearch in terms of its capability to filter out irrelevant code examples from GitHub Code Search and highlight relevant lines of codes.

To evaluate its capability to filter out irrelevant code examples, we count the number of code example files that needs to be investigated to reach the first relevant file in the list of files returned. To count this, we randomly select 11 Android API's queries from Fazzini et al. work [5] as shown in Table I. These queries simulate developers that want to know how to replace deprecated APIs by learning how to use both the deprecated and replacement APIs. On average, we found that 3.1 files need to be investigated to reach the first relevant file if we were to use GitHub Code Search instead of AUSearch while the first file returned by AUSearch is always relevant.

TABLE I
Selected API Queries

API Query
Notification.Builder#addAction(int, CharSequence, PendingIntent)
Notification.Builder#addAction(Action)
TextView#setTextAppearance(Context, int)
Html#fromHtml(String)
Vibrator#vibrate(long)
Vibrator#vibrate(VibrationEffect)
Vibrator#vibrate(VibrationEffect, AudioAttributes)
LocationManager#removeGpsStatusListener(GpsStatus.Listener)
LocationManager#addGpsStatusListener(GpsStatus.Listener)
AudioManager#abandonAudioFocus(OnAudioFocusChangeListener)
AudioManager#requestAudioFocus(OnAudioFocusChangeListener, int, int)

To evaluate its capability to highlight relevant lines of code in a code snippet, we compare the accuracy of the highlighted lines between AUSearch and GitHub Code Search. We define highlight accuracy as follows: $Accuracy = \frac{\#TH}{\#H}$, where $\#H$ is the number of highlighted lines in the code snippet and $\#TH$ is the number of highlighted lines in the code snippet that actually contain the target API usage.

We measure average highlight accuracy by using the same queries as above and compute the highlight accuracy for each combination of query and tool (i.e., AUSearch or GitHub Code Search). We use the first code snippet in AUSearch result for comparison. We found that GitHub Code Search's average highlight accuracy is 0.09 while AUSearch average highlight accuracy is 1.

VI. Discussion

We see that AUSearch is much more accurate than GitHub Code Search because it filters out irrelevant results with type resolution and highlights relevant lines of code.

In the research community, Boa [4] is a popular tool to mine software repositories. However, it cannot be used for the use case of finding API usage, mainly because it does not support type resolution. Without type resolution, Boa would not be able to differentiate methods with the same name and number of parameters but belong to different objects. It would also not be able to differentiate methods with the same name and number of parameters, but the parameters are actually of different types than the ones that we are searching for.

Beside type resolution support, AUSearch is better for API usage than Boa since it integrates directly with GitHub Code Search. GitHub Code Search indexes the most recent repositories in GitHub. Thus, AUSearch can get the most up-to-date result. On the other hand, Boa's repositories are not as up-to-date. In fact, the number of Boa's repositories never increases since 2015.[6] Another disadvantage of using Boa for API usage search is that we need to learn a new programming language before we can actually perform the API usage search. AUSearch is superior in this aspect as we can directly perform the search without a relatively high learning curve. Moreover, Boa is run in a centralized infrastructure while AUSearch can be run anywhere by downloading and running our code – it

[6] http://boa.cs.iastate.edu/stats/index.php

does not require one to download large indexed source code data.

Although AUSearch currently supports only Java programming language, the same idea can be extended to other programming languages. The extension would depend on the existence and the maturity of the type resolution tool in the corresponding programming language.

VII. Related Work

Code search is an active research area in software engineering [1], [6]–[10], [15]. Krugler [9] simply treated code as text and applied web search algorithm for code search. Hill et al. [8] improved code search by reformulating queries according to the position of the query words and the query semantic role. Wang et al. [14] proposed to incorporate user feedback to improve the relevancy of code search results. Bajracharya et al. [1] proposed a code search engine named Sourcerer that combines the texts from the program with the structural information of the program to perform the code search. Lv et al. [10] proposed CodeHow, a code search engine that expands the natural-language query by adding potentially relevant APIs. Similarly, Zhang et al. [15] expands the natural-language query by adding identifiers that are semantically similar with the query. Gu et al. [6] proposed DeepCS, a deep learning model for code search that works by embedding method in programming language and query in natural language together. DeepCS optimizes the similarity of a query embedding with its relevant methods' embeddings.

Different from the above work on code search, our tool targets different use case. Instead of searching for code examples to implement a certain task, our tool targets a use case in which we want to search code examples that contain certain APIs' usages in order to learn from them. Thus, our work complement the above work on code search engine and fill the gap on this use case.

VIII. Conclusion and Future Work

Finding code examples is one of the most common task in software development. To find them, developers can make use of code search engine. However, code search engine usually accepts as input a description of programming task and returns pieces of codes that implement the task. In this work, we are interested in another use case in which developers know what API to look for and want to find code examples for this API. To the best of our knowledge, no existing tool had been designed to address this use case satisfyingly. Existing tools such as GitHub Code Search and Boa [4] may return irrelevant code examples and not always easy to work with. GitHub Code Search may highlight lines of codes that are not related to API usages. On the other hand, Boa requires users to write programs to be able to find API usages.

In this work, we develop AUSearch to overcome the above limitations of existing tools. AUSearch performs type resolution to ensure that it always return only relevant code examples, i.e., code examples that actually contain the API usage that the users are looking for. Using the resolved type, AUSearch is also capable of highlighting lines of codes containing the API usage, making it easier to investigate the returned code examples. In the future, we plan to add supports for other programming languages to AUSearch and use AUSearch for downstream tasks (e.g., automated API usage update, enriching API documentation with code examples, etc.).

References

[1] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[2] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

[3] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.

[4] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[5] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 204–215. ACM, 2019.

[6] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[7] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.

[8] Emily Hill, Lori Pollock, and K Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 524–527. IEEE Computer Society, 2011.

[9] Ken Krugler. Krugle code search architecture. In *Finding Source Code on the Web for Remix and Reuse*, pages 103–120. Springer, 2013.

[10] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE, 2015.

[11] Collin McMillan, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang, and Bamshad Mobasher. Recommending source code for use in rapid software prototypes. In *Proceedings of the 34th International Conference on Software Engineering*, pages 848–858. IEEE Press, 2012.

[12] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201. ACM, 2015.

[13] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*, pages 174–188. IBM Corp., 2010.

[14] Shaowei Wang, David Lo, and Lingxiao Jiang. Active code search: incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 677–682. ACM, 2014.

[15] Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. Expanding queries for code search using semantically related api class-names. *IEEE Transactions on Software Engineering*, 44(11):1070–1082, 2017.