2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)

2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678576

APIzation: Generating Reusable APIs from StackOverflow Code Snippets

Valerio Terragni
University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

Pasquale Salza
University of Zurich
Zurich, Switzerland
salza@ifi.uzh.ch

***Abstract*—Developer forums like StackOverflow have become essential resources to modern software development practices. However, many code snippets lack a well-defined method declaration, and thus they are often incomplete for immediate reuse. Developers must adapt the retrieved code snippets by parameterizing the variables involved and identifying the return value. This activity, which we call APIzation of a code snippet, can be tedious and time-consuming. In this paper, we present APIZATOR to perform APIzations of JAVA code snippets automatically. APIZATOR is grounded by four common patterns that we extracted by studying real APIzations in GitHub. APIZATOR presents a static analysis algorithm that automatically extracts the method parameters and return statements. We evaluated APIZATOR with a ground-truth of 200 APIzations collected from 20 developers. For 113 (56.50 %) and 115 (57.50 %) APIzations, APIZATOR and the developers extracted identical parameters and return statements, respectively. For 163 (81.50 %) APIzations, either the parameters or the return statements were identical.**

***Index Terms*—APIs, software reuse, code snippets, StackOverflow, GitHub, program analysis, program synthesis**

I. Introduction

Developers' Q&A websites, such as StackOverflow (SO), have gained a lot of popularity. These websites contain millions of crowd-curated code snippets that represent solutions to various programming tasks. These code snippets are extremely useful to both developers and researchers. Developers often search for them to draw inspiration or simply reuse them [1], [2], [3]. Researchers often rely on SO to accomplish various software engineering goals [4].

When reusing SO code snippets, developers and researchers face a major obstacle: most SO code snippets do not compile [5], [6], [7]. It mainly occurs because they are written for illustrative purposes, to convey solutions at a high level, without implementation details [8]. Terragni et al. have shown that ≈92 % of 491,906 analyzed SO code snippets are uncompilable [5]. A common missing implementation detail is the type declaration [5], [6]. For instance, the JAVA SO code snippet in Fig. 1 (left side) misses the declaration of type `Calendar` and `Date`. Researchers have tackled this issue by proposing techniques to identify the import declarations required to compile SO code snippets [5], [9].

Another common missing detail in SO code snippets is a well-formed method declaration that defines the parameters (input) and return statements (output) [10], [5]. Terragni et al. have shown that ≈56 % of JAVA SO code snippets constitute *dangling statements*, which are not embedded in any class or method declarations [5]. The SO code snippet in Fig. 1 (left side) is an example of dangling statements. One could automatically wrap the code snippet inside a generic method declaration [5], [6] (e.g., the `main` function). It would resolve compilation errors but would not recover the proper method declaration that exposes the intended input and output of the code snippet. The absence of a proper interface prevents the direct reuse of SO code snippets. Thus, some manual effort is required to identify the inputs and outputs of the code snippets.

We use the term *"APIzation"* to indicate the activity of creating an Application Program Interface (API) for those SO code snippets without a well-formed method declaration. Fig. 1 (center) shows a manual APIzation of a SO code snippet.

In this paper, we study the automatic APIzation of JAVA SO code snippets, which would bring important benefits. Developers would reduce the effort of integrating SO code snippets into their codebases, which is known to be a tedious and time-consuming activity [11]. Given an automatically generated API of a SO code snippet, developers can either copy and paste the API in the codebase or incorporate the method body of the API inside an existing method. The presence of an API facilitates the latter option. Indeed, an API explicitly shows the input and output of the code snippet, which helps to both understand and incorporate the SO code. Moreover, the automatic APIzation SO code snippets can lead to a large catalog of code samples with well-defined interfaces, providing value for both developers and researchers.

Towards these goals, we conducted an investigatory study to understand how developers perform APIzations from SO code snippets to JAVA methods found in GitHub (GH). The insights gained from this study led to four common APIzation patterns to extract method parameters and return statements. Grounded by these patterns, we propose a technique called APIZATOR for the automated APIzation of SO code snippets. To the best of our knowledge, APIZATOR is the first technique of its kind. APIZATOR statically analyzes a given code snippet to find matches for the four patterns. If it finds matches, APIZATOR extracts the parameters and return statements and outputs a compilable API. For completeness, APIZATOR uses a Part-of-Speech (POS) Tagger to generate a method name from the SO question title, and creates a JAVADOC containing the title and link of the corresponding SO page.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00055

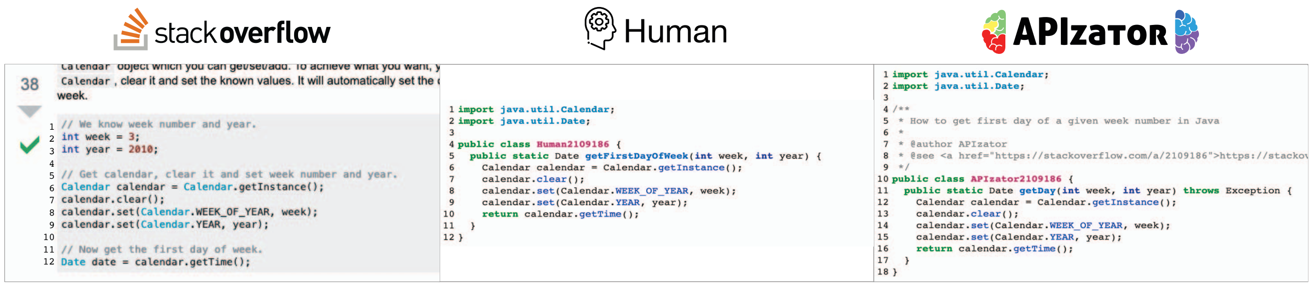

Fig. 1. APIzation of a SO code snippet. APIZATOR and the human produced identical APIs (except for the method name and JAVADOC).

We evaluated APIZATOR with a ground truth of 200 APIzations performed by 20 human participants, obtaining 200 pairs of human- and tool-produced APIs. We compared each pair to assess the effectiveness of APIZATOR. For 113 (56.50 %) and 115 (57.50 %) API pairs the parameter list and return statements are identical, respectively. For 163 (81.50 %) APIs generated by APIZATOR either the return statements or the method parameters are identical to those produced by the developers. For instance, Fig. 1 (right side) shows the API produced by APIZATOR, which is identical to the one created by the developer (excluding the method name and JAVADOC).

To demonstrate one of the possible usage scenarios of APIZATOR, we release a search engine at the address https://apization.netlify.app/search/ and as part of our replication package [12]. The users can search for SO code snippets with a natural language query as they would do with a standard search engine. The search results show the SO page as well as its API automatically generated by APIZATOR.

To summarize, the main contributions of this paper are:

- studying the problem of automatically transforming SO code snippets into APIs;
- analyzing real APIzations across SO and GH projects, extracting four common APIzation patterns;
- proposing a technique called APIZATOR to transform SO code snippets into well-formed JAVA method declarations;
- evaluating APIZATOR against a ground truth of 200 APIzations performed by 20 JAVA developers;
- releasing at the address https://apization.netlify.app all the experimental data;
- releasing 109,930 APIs automatically extracted from SO code snippets, which could power SO-centric research.

II. PRELIMINARIES AND PROBLEM DEFINITION

In this paper, we target JAVA code snippets found in StackOverflow (SO), the most popular Q&A website for developers [13]. The process of APIzation takes in input a SO code snippet and generates a JAVA method declaration. We now describe in detail the input and output of such a process.

Input: A JAVA code snippet from SO. A SO page is composed of a question post and a series of answer posts. Each question post contains a title, a series of tags, and a description. A post can contain one or more code snippets. A **Code Snippet (CS)** is an ordered sequence of source code lines.

Output: A compilable and well-formed JAVA method declaration that defines the code snippet in input. A *method declaration*, which we call Application Program Interface (API), consists of the following six attributes: (i) *modifiers*, which set the access level (e.g., `public`), or achieve specific functionalities (e.g., `static`); (ii) *return type*, which indicates the type of value that the method returns (`void` if none); (iii) *method name*, which describes the behavior of the method; (iv) *parameter list*, which specifies the types and identifiers of the method arguments; (v) *throws clause*, which declares any checked exception classes that the method body can throw; (vi) *method body*, which contains the statements that implement the method.

The *method body* of a *well-formed* API references each of the parameters and contains, if the return type is not `void`, one or more return statements. To make an API compilable, it has to be declared inside a class (e.g., `Human2109186` in Fig. 1) that contains the required *import declarations* (*imports* in short) (e.g., `java.util.Calendar` and `java.util.Date` in Fig. 1). At each class is associated a *classpath* to the library JARs that declare the types in *imports* (e.g., JDK in Fig. 1).

Most JAVA code snippets from SO are composed of dangling statements not enclosed in any method declaration [5], [14], [6] (see Fig. 1). The **process of APIzation** aims at generating well-formed method declarations for such code snippets. It achieves this by performing six actions:

1) choose a method name, e.g., `getFirstDayOfWeek` in Fig. 1;
2) recover missing declarations of variables or types from the code snippet, e.g., `Calendar` and `Date` in Fig. 1;
3) identify which variables in the snippet are the intended input parameters, e.g., variables `week` and `year` in Fig. 1;
4) remove the declarations of such variable from the code snippet, e.g., `int week = 3;` in Fig. 1;
5) infer the output of the snippet, if any, and add a return statement for it, e.g., `return calendar.getTime()` in Fig. 1;
6) enclose the resulting statements in a method declaration with proper parameters and return type, e.g., `public static Date (int week, int year)` in Fig. 1.

Problem definition: Given a JAVA code snippet, the process of **APIzation** generates a compilable and well-formed method declaration for the given code snippet.

III. Understanding Real World APIzations

This section presents an investigatory study to understand how developers perform APIzations. The insights gained from this study led to four common APIzation patterns that establish the foundations of our proposed technique. To collect manual APIzations of StackOverflow (SO) code snippets, we relied on GitHub (GH). Our goal is to find code reuses across SO code snippets and GH projects that represent APIzations. Fig. 2 gives two examples of such manual APIzations. We release the data of our investigatory study in our replication package, published at https://apization.netlify.app/study/.

A. Data Collection

Researchers have experimented two main approaches to identify code reuses across SO and GH [15], [16], [17], [18], [19], [20], [19]: (i) search for explicit SO web links in GH code comments or Javadoc; (ii) search for code clones between SO code snippets and GH code.

Both of these approaches have pros and cons. Relying only on explicit SO web links likely misses many code reuses. In fact, GH developers often do not give proper credit when reusing SO code snippets [18], [20]. It can also lead to spurious code reuses as GH developers may cite a SO post because it discusses a particular issue, which is unrelated to code reuse [20]. Relying only on code clones has the advantage to identify code reuses even without (rare) explicit SO links. However, code clones cannot guarantee that the GH developers performed the APIzation from SO [20], [15].

Because of the complementarity of these two approaches, we decided to consider those code reuses that are identified by both approaches. We will probably miss many code reuses, but we are more confident that the identified ones are genuine. Thus, our goal is to identify pairs $\langle CS, API\rangle$ (where *CS* is a SO code snippet and *API* a GH method) that satisfy all of these three criteria: (i) the comments or Javadoc of API have an explicit link to the SO page containing CS; (ii) API and CS are code clones; (iii) API is an APIzation of CS. We now describe in more detail how we identified such pairs.

Find candidate pairs. We queried the latest snapshot of GH on Google BigQuery [21], which contains ≈1 million projects with the tag Java. We identified 29,035 unique Java files containing explicit links to SO pages. From the retrieved Java files, we identified all the GH methods (API) containing the explicit SO link as a code comment or in the JavaDoc. For each SO link, we extracted the corresponding SO code snippet(s) by querying the latest SO dump. We then pruned all those pairs in which *CS* already contains a well-formed method declaration, or *CS* has less than six lines.

Code clone detection. For each candidate pairs $\langle CS, API\rangle$, we searched for *TYPE 3* code clones [22], i.e., syntactically similar code with inserted, deleted, or updated statements. We chose TYPE 3 clones because both *TYPE 2* and *TYPE 4* are inadequate for our purposes. *TYPE 2* clones require syntactically equivalent code (the only allowed variations are in identifiers, types, whitespace, layout, and comments). This is too restrictive because the APIzations often create APIs with fewer or more statements than the SO code snippets. For example, the human APIzation of Fig. 1 deletes the SO lines 2 and 3 and updates line 12. TYPE 4 clones allow semantically equivalent but syntactically different code. This is too permissive because we are only interested in explicit code reuses.

To detect TYPE 3 clones, we automatically perform *alpha-renaming* of the variables (e.g., `int a = 5` becomes `int int0 = 5`). If there are multiple variables with the same type, we use a progressive id as a suffix. For example, `int a = 5; int b = 10` becomes `int int0 = 5; int int1 = 10`. We also removed comments, new lines, and formatting characters. We treated a pair $\langle CS, API\rangle$ as a TYPE 3 code clone if at least $70\,\%$ of *CS* source code lines are contained in API (we opted for $70\,\%$ following Zhang et al. [19]). This resulted in 330 code clone pairs, referring to 199 unique SO answer posts.

Note that TYPE 3 code clone detection excludes by default TYPE 1 and TYPE 2 clones as they require a $100\,\%$ similar code. This is impossible in our case since APIs always contain a method signature, while the considered code snippets do not.

Manual check. We manually checked each of the 330 code clone pairs to prune those in which the APIs do not represent the APIzation of CS. We pruned the pairs that were spurious code clones (the matched lines were mostly common lines of code such as `try{` and `catch()`). We pruned the pairs that were valid clones, but *CS* was incorporated inside the GH method. These pairs are not APIzations because the GH method declaration does not strictly relate to the SO code snippet.

B. Analysis of the Results

We manually analyzed the retained **135 pairs** to study the variables in the SO code snippet that became method parameters or return statements in the GH method. We followed a coding process inspired by *grounded theory* [23], which derives new theories and concepts by analyzing the data.

We distributed between the two of us the 135 pairs of the SO snippet and matching GH method. For convenience, we used a diff tool to generate a visual representation of the code differences between the snippet and method. Such a representation helped us to quickly identify the APIzation activity performed by the developers. During the *open coding* stage, we analyzed each of the assigned pairs to give a distinct *code* for each of the observed phenomena, i.e., APIzations. In particular, the question that drove the open coding was: "What are the characteristics of the variables in the SO snippet that became parameters and return statements in the GH method?" Examples of produced codes are: "undeclared variable", "the variable has a constant value", and "the variable is used as an argument in a `System.out.println` invocation."

Then, we refined the codes by grouping similar concepts and finding connections between them, i.e., *axial coding*. Then, we concluded the patterns' identification with *selective coding*.

Each of the authors independently analyzed the pairs and eventually discussed the results to reach a consensus. Finally, we identified four common patterns (`PATT-notdecl`, `PATT-`

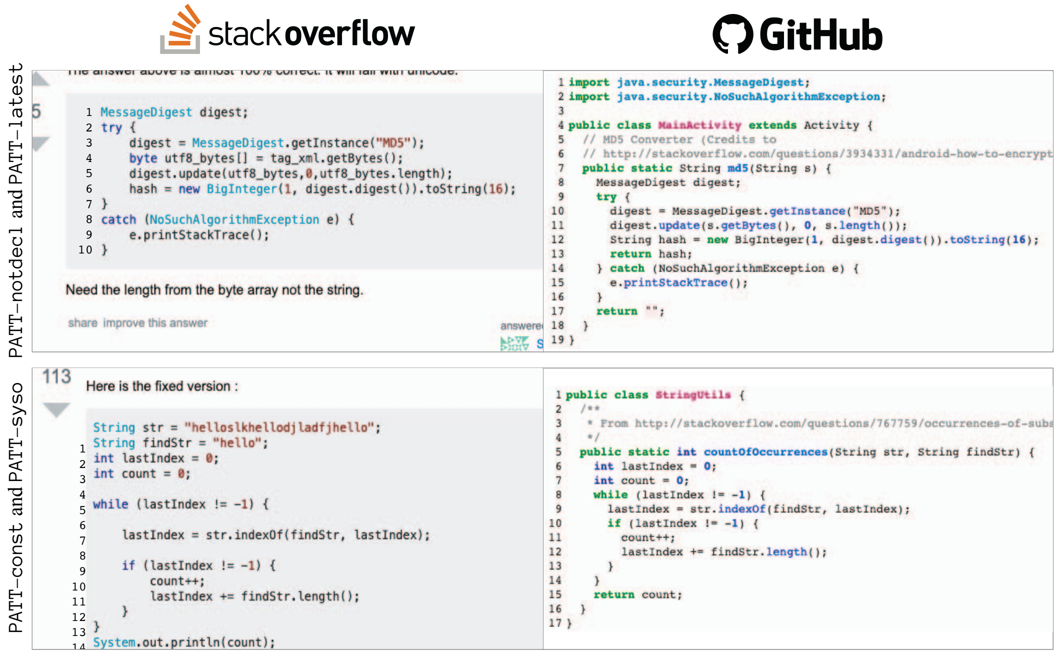

Fig. 2. Examples of APIzation patterns found in APIzations from StackOverflow to GitHub.

`const`, `PATT-latest`, and `PATT-syso`) that characterize and define general APIzation activities.

1) Method Parameters: The 135 SO code snippets reference 509 variables with an average of 3.77 variables per code snippet. Among these 509 variables, 45 became method parameters in the corresponding GH method. Among these 45 variables, 32 (71.11 %) match `PATT-notdecl` and 9 (20.00 %) match `PATT-const`. For the remaining four variables, we were not able to generalize any pattern.

`PATT-notdecl.` *A variable v that is referenced in CS is extracted as a parameter if CS lacks the declaration of v.*

Fig. 2 (top) shows one of the analyzed pairs that exhibits such a pattern (the SO code snippet *(CS)* is on the left and the GH method (*API*) on the right). The line 4 of the CS references an undeclared variable `tag_xml`, and the GH developer extracted `tag_xml` as a method parameter at line 7 (renaming it to `s`). A possible rationale for this pattern is that undeclared variables in SO code snippets are commonly intended as the (implicit) inputs of a programming task.

`PATT-const.` *A variable v declared in CS is extracted as a parameter if (i) CS initializes v with a hard-coded value; and (ii) CS does not have loops that modify the value of v.*

Fig. 2 (bottom) shows a pair that manifests such a pattern. The SO code snippet declares four variables: `str`, `findStr`, `lastIndex`, and `count`. It initializes them with hard-coded values that embed data directly into the source code. These four variables match criterion (i), but only `str` and `findStr` match also criterion (ii). In fact, only `str` and `findStr` became method parameters in the GH method. The variables `lastIndex` and `count` are excluded because the SO while loop can modify their values. Extracting such variables would change the semantics of the while loop. For example, if `count` is extracted as a parameter, a user can invoke the API with a `count` value different from zero, making the API return a meaningless value. A possible rationale for this pattern is that SO code snippets often exemplify programming tasks, and thus the hard-coded values represent a particular instance of the inputs.

2) Return Statements: Among the 135 GH methods, 63 (46.67 %) lack return statement(s) (the return type is `void`) and 72 (53.33 %) have return statement(s). Among such 72 GH methods, 31 (43.06 %) match `PATT-latest`, and 6 (8.33 %) match `PATT-syso`. For the remaining methods, we could not generalize any pattern or the SO code snippet already contained return statement(s).

`PATT-latest.` *The assignment of a variable in CS becomes the return statement if it is the last statement in CS.*

For example, the SO snippet in Fig. 2 (top) ends with the assignment of the `hash` variable (we ignored exception handling as last statements because they are unrelated to the semantics of the code snippet), and the GH method returns `hash` of type `String`. Intuitively, the last statement of a SO snippet often characterizes its output. Indeed, it is unlikely that developers end the snippet with a value irrelevant to the final intent of the programming task.

`PATT-syso.` *If the last statement in CS is a `System.out.println` call, its argument becomes the return statement.*

An example of such a pattern is the SO snippet in Fig. 2 (bottom). The code snippet ends with `System.out.println(count)`, and the GH method returns `count` of type `int`. Because SO users write code snippets for illustration purposes, they often add a print of the output value to show the result when the snippet is being executed.

3) Manual Application of the Patterns: After identifying the four patterns, we applied them to the whole dataset to evaluate if they lead to spurious parameters and return statements. Among the 464 SO variables that did not become parameters in the corresponding GH methods, 14 ($3.02\,\%$) and 8 ($1.72\,\%$) variables match `PATT-notdecl` and `PATT-const`, respectively. Among the 93 GH methods in which we did not identify any pattern or lack return statements (i.e., return type is `void`), the patterns `PATT-latest` and `PATT-syso` match 4 ($4.30\,\%$) and 1 ($1.08\,\%$) variables, respectively.

This indicates that the four patterns lead to a few spurious parameters and return statements. Thus, finding matches of these patterns in SO code snippets is a viable solution for automating the APIzation process.

IV. APIZATOR

This paper presents APIZATOR to automatically transform JAVA SO code snippets into reusable and compilable APIs. Algorithm 1 describes the process of APIZATOR in detail.

Algorithm 1: APIZATOR

```
input  : CS = ⟨s_1, ..., s_n⟩, a SO code snippet
         SO-page, the SO page of CS
         JARs, a set of external libraries
output : API, a method declaration for CS
         imports, the import declarations for API
         classpath, the classpath for API
1  ⟨imports, classpath⟩ ← GetOrDefault(CS, JARs)
2  if CS is a well-formed method declaration (CS ≡ API) then
3  |  return ⟨API, imports, classpath⟩
4  API ← CreateInitialMethodDeclaration(imports, CS)
5  API.method-name ← CreateMethodName(SO-page)
6  while Compile(API, imports, classpath) → errors ≠ ∅ do
7  |  if errors ⊆ missing-type-decl then
8  |  |  ⟨imports, classpath⟩ ← CSnippEx(errors, JARs, imports,
   |  |    classpath)
   |  /*                     PATT-notdecl                      */
9  |  else if errors ⊆ missing-variable-decl then
10 |  |  for v ∈ (errors ∩ missing-variable-decl) do
11 |  |  |  ⟨τ, imports, classpath⟩ ← RecoverVarType(v, API,
   |  |  |    JARs, imports, classpath)
12 |  |  |  T[v] ← τ
13 |  |  |  add ⟨τ, v⟩ to API.parameter-list
14 |  else return ∅
/*                          PATT-const                          */
15 LP-VARS ← GetLoopChangingVars(API.method-body)
16 for s_i ∈ API.method-body do
17 |  case s_i : τ v = ϵ do     // Variable decl. and init.
18 |  |  T[v] ← τ
19 |  |  add v to ALREADY-INIT-VARS
20 |  |  if IsHardCoded(τ, ϵ) ∧ v ∉ LP-VARS then
21 |  |  |  add ⟨τ, v⟩ to API.parameter-list
22 |  |  |  remove s_i from API.method-body
23 |  case s_i : τ v do                // Variable declaration
24 |  |  ⟨T[v], S[v]⟩ ← ⟨τ, s_i⟩
25 |  case s_i : v = ϵ do               // Variable assignment
26 |  |  if v ∉ ALREADY-INIT-VARS then
27 |  |  |  add v to ALREADY-INIT-VARS
28 |  |  |  if IsHardCoded(τ, ϵ) ∧ v ∉ LP-VARS then
29 |  |  |  |  add ⟨T[τ], v⟩ to API.parameter-list
30 |  |  |  |  remove s_i from API.method-body
31 |  |  |  |  remove S[v] from API.method-body
/*                          PATT-latest                         */
32 case s_n : τ v = ϵ do          // Variable decl. and init.
33 |  API.return-type ← τ
34 |  replace s_n in API.method-body with return ϵ;
35 case s_n : v = ϵ do                  // Variable assignment
36 |  API.return-type ← T[v]
37 |  replace s_n in API.method-body with return ϵ;
/*                          PATT-syso                           */
38 case s_n : System.out.println(string-literal + ϵ) ∨
     System.out.println(ϵ) do
39 |  API.return-type ← GetTypeOfExp(ϵ, imports, classpath)
40 |  replace s_n in API.method-body with return ϵ;
41 otherwise do
42 |  API.return-type ← void
43 return ⟨API, imports, classpath⟩
```

Input and output. APIZATOR takes as an input: (i) *CS*, a SO code snippet; (ii) *SO-page*, the SO page of the snippet, which APIZATOR uses to generate the method name; (iii) *JARs*, a set of common JAVA libraries to recover the missing import and variable declarations [5]. APIZATOR outputs (i) *API*, the method declaration of *CS*; (ii) *imports*, the import declarations of the non-primitive types that *API* references; (iii) *classpath*, the libraries in *JARs* that declare the types in *imports*.

Preliminary check (Lines 1 to 3). Algorithm 1 starts by checking if *CS* already contains import declarations (Line 1). If yes, it extracts them and searches in *JARs* for the corresponding libraries, which it adds to the *classpath*. If not (the common case), it creates an empty *imports* list and an initial *classpath* with only the JDK JAR library. Next, it checks if *CS* already defines a well-formed and compilable API. If so, it returns *CS*, *imports*, and *classpath* (Lines 2 to 3), otherwise it starts the "APIzation" process.

Initialization of the API (Line 4). The "APIzation" process begins by initializing *API*, the method declaration for *CS*. By default, the *modifiers* of *API* are `public` (because APIs must be accessible by any other class) and `static` (to avoid instantiating objects for invoking the API). The *throws-clause* of *API* is the generic `java.lang.Exception`. APIZATOR initializes the *method-body* of *API* with *CS*, the *return type* with `void` and the *parameter list* with the empty list.

Method name generation (Line 5). For completeness, APIZATOR generates a method name for the API from the title of the SO page associated with the code snippet [24]. Indeed, the title of the SO page often summarizes the intent of the programming task. APIZATOR relies on a Part-of-Speech (POS) Tagger [25] to assign parts of speech (e.g., nouns, verbs, and adjectives) to each word in the title. Then, APIZATOR creates the method name by combining the main "verb" of the sentence and the corresponding "direct object" (i.e., noun). We consider these two parts of speech because method names are typically verbs or verb phrases. We do not claim this to be a contribution to this work. In the future, we plan to investigate state-of-the-art approaches for generating method names [26].

For a statically-typed programming language such as JAVA, type inference is precise and unambiguous only with compilable code [27]. APIZATOR requires complete type information to know the type of the method parameters and return statements. However, assuming only compilable code is infeasible because most SO code snippets do not compile [5], [6], [9]. Line 6 of Algorithm 1 tries to compile the *API* (wrapping it in a synthetic JAVA class) with the current *imports* and *classpath*. If any compilation errors arise, APIZATOR attempts to fix them. Note that, APIZATOR needs to re-compile *API* iteratively because fixing a compilation error may reveal others [5]. APIZATOR supports two types of compilation errors: (i) missing type declarations (Line 7) and (ii) missing variable declarations (Line 9). For other error types APIZATOR terminates (Line 14).

Recover missing type declarations (Lines 7 to 8). APIZATOR relies on CSNIPPEX [5] to fix missing type declarations. CSNIPPEX recovers the import declarations that fix such errors by querying the fully-qualified names of the classes declared in *JARs*. This is challenging because there are often many fully qualified names with the same simple name. CSNIPPEX addresses the challenge with a greedy algorithm based on the *clustering hypothesis*: *"the referred library classes in a* JAVA *source file often come from the same libraries, and hence their import declarations tend to form clusters that share common package names"* [5]. For example, the code snippet in Fig. 2 (top) leads to two missing type declarations: `MessageDigest` and `NoSuchAlgorithmException`. CSNIPPEX identifies the correct import declarations because they share the same package name `java.security`. CSNIPPEX adds the corresponding JAVA libraries in the classpath and leverages the feedback of the compiler to check if the errors are fixed.

Recover missing variable declarations (`PATT-notdecl`, Lines 9 to 13). APIZATOR recovers missing variable declarations to fix the compilation errors and to find matches of `PATT-notdecl`, which considers undeclared variables as method parameters. To recover missing variable declarations, APIZATOR relies on the RECOVERVARTYPE function (Line 11). Given an API with an undeclared variable v, this function identifies the most plausible type of v by leveraging the usages of v in the API, which follows the BAKER approach [28].

For example, the SO code snippet in Fig. 2 (top) lacks the declaration of variable `tag_xml`. APIZATOR correctly infers that the type of `tag_xml` is `java.lang.String` because (i) the code snippet invokes the method `public byte[] getBytes()` using `tag_xml` as the object receiver, and (ii) `java.lang.String` declares a method with the same name and return type. When there are multiple plausible types, APIZATOR uses a successful compilation as a proxy for correctness. In fact, *API* compiles without errors if the declaration of `tag_xml` has type `java.lang.String`. Line 11 of Algorithm 1 also updates *imports* and *classpath* accordingly, which remain unchanged in our example (the package `java.lang` is imported by default). Next, APIZATOR updates the map $\mathcal{T}$, which stores for each declared variable in *CS* its type. Line 13 of Algorithm 1 adds `tag_xml` as a parameter. This is the correct parameter, as it was also used by the GH developer that performed the manual APIzation (`tag_xml` is renamed to `s`).

Recognize hard-coded initializations (`PATT-const`, Lines 15 to 31). Function GETLOOPCHANGINGVARS returns the variables *LP-VARS* in the method body that have at least one assignment inside a loop (Line 15 of Algorithm 1). `PATT-const` needs to identify such variables because they will not be considered as parameters. Line 16 of Algorithm 1 scans the statements in *API.method-body* to search for variable initializations that meet the conditions of `PATT-const`. The scan considers the following three statements types:

1) Variable declaration and initialization $\tau\ v = \epsilon$. For example, `String findString = "hello"` in Fig. 2 ($\tau =$ `String`, $v =$ `findString`, and $\epsilon =$ `"hello"`). When APIZATOR encounters such statements, it maps τ to v, and it adds v to *ALREADY-INIT-VARS*, which is a set that maintains the variables that are already initialized. The function ISHARDCODED takes in input the type τ and the expression ϵ and it returns `true` if ϵ is a hard-coded value, `false` otherwise.

If τ is primitive or `String`, the function returns `true` if ϵ does not contain identifiers (i.e., variable, class, method names), `false` otherwise. Identifiers characterize data dependencies. For example, ISHARDCODED(`String`, `"hello"`) returns `true` because `"hello"` does not contain identifiers.

As another example, consider the following code snippet.

```
String a = "world";
String b = "hello" + a;
```

ISHARDCODED (`String`, `"hello" + a`) returns `false` because ϵ = `"hello" + a` is data dependent to the variable `a`.

If τ is non-primitive, ϵ must always contain at least one identifier (`null` is also an identifier). For example the ϵ of the statement `Calendar calendar = Calendar.getInstance();` in Fig. 1 has `Calendar` and `getInstance` as identifiers. As such, for non-primitive types, ISHARDCODED returns `true` if τ is a subclass of `java.util.Collection` and after the statement s_i follow $n > 1$ statements that add elements to the collection (e.g., invoke `add` methods for `java.util.List`, and `put` methods for `java.util.Map`). APIZATOR makes a similar consideration for matrices and arrays.

Line 20 Algorithm 1 checks if the variable v meets both `PATT-const` criteria (v is initialized with a hard coded value and is not a loop variable). If yes, it adds v of type τ to the parameter list and removes the declaration statement s_i from the method body. For example, the statement `String findStr = "hello"` at Line 2 in Fig. 2 (bottom) meets both requirements, and thus APIZATOR makes `findStr` a method parameter and removes the statement.

2) Variable declaration $\tau\ v$. These statements are only declarations without initializations. For such statements, APIZATOR saves the type τ of v and statement s_i. APIZATOR needs this information if later it encounters the initialization of v.

3) Variable assignment $v = \epsilon$. At Line 26, Algorithm 1 checks if v belongs to *ALREADY-INIT-VARS*. If yes, APIZATOR skips the statement because it already encountered the initialization of v. If not, APIZATOR has found the initialization of v. Then, it updates *ALREADY-INIT-VARS* and checks if the `PATT-const` criteria are met. If yes, it recovers the type of v

from $\mathcal{T}$ and adds the v to the parameter list. Then, it removes from the method body both the statement that declares v ($\mathcal{S}[v]$) and the statement that initializes v (s_i).

Check the last statement (`PATT-latest`, and `PATT-syso`, Lines 32 to 43). At Lines 32 to 43, Algorithm 1 analyzes the last statement (s_n) to decide whether it should be considered as the return statement.

If s_n is a variable declaration or an assignment, then s_n matches `PATT-latest`, and thus APIZATOR replaces s_n with a statement that returns the expression ϵ. APIZATOR recovers the type of ϵ directly from s_n (if s_n is a declaration) or from $\mathcal{T}$ (if s_n is an assignment).

If s_n is an invocation to `System.out.println`, then s_n matches `PATT-syso`. Algorithm 1 extracts the argument ϵ of the invocation by removing the first string-literal (if it exists), which is likely to represent a placeholder (e.g., `System.out.println("result :" + s)`). Given ϵ, Algorithm 1 recovers τ, the type of ϵ, which will be the return type of *API*. Although `System.out.println` handles String objects, τ is not necessarily `String`. In fact, `System.out.println(object)` invokes that object's `toString()` method to convert the object to a `String` representation. For example, given the last statement `System.out.println(count)` in Fig. 2 (bottom), the return type should be `int` and not `String`. The function GETTYPEOFEXP analyses ϵ and *classpath* to recover τ. If ϵ is a variable v, the function recovers τ from the map $\mathcal{T}[v]$. If ϵ is a method invocation m, the function consults the declaration of m in *classpath* to get its return type.

V. EVALUATION

This section discusses a series of experiments that we conducted to evaluate APIZATOR. In the context of our study, we formulated the following three research questions:

- **RQ1** *Does* APIZATOR *generate APIs that are **identical** to the ones that a human would produce?*
- **RQ2** *How effective the* APIZATOR *algorithm is in identifying the **method parameters**?*
- **RQ3** *How effective the* APIZATOR *algorithm is in identifying the **return statements**?*

To answer these research questions, we collected a ground truth of human-produced APIs. We decided not to rely on the GitHub (GH) dataset used in Section III to avoid overfitting (APIZATOR is based on the insights extracted from the GH dataset). Instead, we asked 20 *human participants to build a ground-truth of* 200 *APIs* by manually performing the APIzation of 200 SO code snippets. All the evaluation data is available in our replication package [12] and published at https://apization.netlify.app/evaluation/.

A. Evaluation Setup

1) Creating a Collection of APIs from StackOverflow: We considered the SO data dump of May 2019 [29], which contains 1,014,980 SO pages with the tag JAVA. From these SO pages, we selected all the 1,730,251 SO answer posts with at least one code snippet.

Identifying the compilable SO code snippets. We first ran CSNIPPEX on each of the 1,730,251 SO answer posts, to identify those code snippets for which CSNIPPEX is able to recover the missing type declarations. CSNIPPEX requires a set of common JAVA libraries JARs as an input [5]. We obtained such a set by downloading the latest JAR of the top three libraries of each category in the MAVEN REPOSITORY [30]. We then used the dependency resolver of MAVEN to identify the additional JARs that belong to the runtime dependencies of the selected libraries. In total, we obtained 748 JAR files. Running CSNIPPEX with a time-budget of 5 seconds for each post, it returned compilable JAVA files for 141,064 SO posts.

Creating the SO APIs. We ran APIZATOR on these 141,064 SO answer posts with a time budget of 10 seconds each, obtaining **109,930 APIs**. APIZATOR skipped 31,134 out of the 141,064 posts because the APIzation is either impossible or ambiguous. It is impossible for abstract methods and for JAVA files with only field or class declarations. It is ambiguous for files that have more than one public method or that declare more than one class. In such cases, APIZATOR cannot infer which public method is the intended API.

It is worth noting that, for each of the produced APIs, APIZATOR generates a JAVADOC containing the link to the original SO post from which the code was taken (see Fig. 1). This is compliant with the SO Terms of Service, which, at present, states that user contributions are licensed under *Creative Commons Attribution-ShareAlike*[1]. The specific license terms depend on the date of publication of the SO post, but all of them require appropriate credit to the authors of the content, i.e., a link to the SO post. In fact, the *CC BY-SA* license allows re-distribution and re-use of a licensed work (even for commercial use) on the condition that the creator is appropriately credited. However, it is the responsibility of the end user to keep the link of the SO post associated with the APIZATOR-generated APIs. Similarly, manually copying and adapting a SO snippet should require appropriate credit by including a link to the SO post [18].

2) Selecting the APIs for the Evaluation: From the 109,930 APIs we selected those that satisfy five properties:

I. The SO page of the API is a "how to" question. Following previous SO studies, we assume that the most useful code snippets are in answers to "how to" questions [31], [10]. We identified such questions by the presence of the word "how" in the SO page title [31].

II. The SO post associated with the API is the accepted answer or has a score of at least two (two is the average score in SO). This is to select high-quality code snippets.

III. The SO post associated with the API contains exactly one code snippet. This is to avoid ambiguity, as multiple code snippets in the same SO post often refer to alternative solutions of the same programming task. Having only one code snippet, the human participant does not need to decide which one to consider.

[1] https://stackoverflow.com/legal/terms-of-service/public#licensing

TABLE I
RQ2 ANALYSIS AND COMPARISON OF THE HUMAN- (P_{H}) AND APIZATOR-PRODUCED (P_{A}) PARAMETER LISTS

Param. $\|P_{\mathrm{H}}\|$	Human APIs	$P_{\mathrm{H}} \equiv P_{\mathrm{A}}$		$\|P_{\mathrm{H}} \setminus P_{\mathrm{A}}\|$				$\|P_{\mathrm{H}} \cap P_{\mathrm{A}}\|$				$\|P_{\mathrm{A}} \setminus P_{\mathrm{H}}\|$				Jaccard Distance (JD)			
		Count	%	Mean	Min	Mdn	Max	Mean	Min	Mdn	Max	Mean	Min	Mdn	Max	Mean	Min	Mdn	Max
0	58	45	77.59	–	–	–	–	–	–	–	–	0.36	0.00	0.00	5.00	0.22	0.00	0.00	1.00
1	93	60	64.52	0.32	0.00	0.00	1.00	0.68	0.00	1.00	1.00	0.13	0.00	0.00	2.00	0.34	0.00	0.00	1.00
2	35	7	20.00	1.14	0.00	1.00	2.00	0.86	0.00	1.00	2.00	0.29	0.00	0.00	2.00	0.58	0.00	0.50	1.00
$\geq$3	14	1	7.14	2.86	0.00	3.00	6.00	0.64	0.00	0.00	4.00	0.21	0.00	0.00	1.00	0.82	0.00	1.00	1.00
Total ($\geq$ 0)	200	113	56.50	0.77	0.00	0.50	6.00	0.72	0.00	1.00	4.00	0.23	0.00	0.00	5.00	0.38	0.00	0.00	1.00

IV. The import declarations of the API do not refer to any external libraries other than the JDK. Participants might produce incorrect APIzations, for instance, if they are unfamiliar with a particular library.

V. The SO code snippet associated with the API *does not* contain a well-formed method declaration. In such cases, the code snippet is already an API, and Algorithm 1 has no effect.

A total of 9,901 APIs satisfy all of these properties. We sorted them by the view count of the corresponding SO post and selected the first 200 APIs. It is worth noting that we had to manually discard some of the APIs in which the APIzation is not a reasonable operation (even though the above-mentioned properties were satisfied). For example, when the SO code snippet is not a programming task (e.g., it shows usage examples of JDK classes), or it is semantically incomplete (e.g., it contains placeholders for missing functionality). The 200 APIs have 11.45 lines of code on average. The corresponding SO posts have an average number of views of ≈66,000, and an average score of 46.62.

3) Ground-Truth of Human APIzations: We partitioned the 200 code snippets in 200 disjoint sets and sent them to 20 expert JAVA developers in the authors' circle of acquaintances. Each participant had assigned ten SO posts. The 20 participants come from seven different countries and constitute a heterogeneous group of ten Ph.D. students majoring in software engineering, five senior software engineering researchers, and five professional JAVA developers. The participants have several years of experience in JAVA programming: 9.8 years on average (min 1, median 9.5, and max 15). None of the 20 participants knew that APIZATOR exists and how it generates APIs. Thus, they performed the manual APIzation without biases.

Experiment description. Each participant received a script that interacts via the command line. The script gives the instructions and monitors the APIzation time. It was an uncontrolled experiment, thus they ran the script at their convenient time. We decided to avoid guidelines to let the participants decide what APIzation means to them. Instead, the script exemplifies the concept with an example. After showing the example, the script shows the SO page of the first assigned code snippet. It then asks the participant to read the SO page to understand the semantics of the code snippet, and to write in the IDE a method declaration for it. This process repeats until the participant completes the ten assigned code snippets. This led to 200 pairs $\langle API_{\mathrm{H}}, API_{\mathrm{A}}\rangle$ of human- (API_{H}) and APIZATOR-produced (API_{A}) APIs from the same SO code snippet. We release the instructions of the script in our replication package [12] and published at https://apization.netlify.app/evaluation/script/.

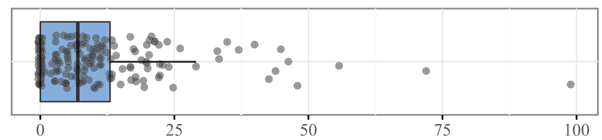

Fig. 3. Distribution of the number of AST differences.

Pre-processing the human APIs. Before comparing the pairs, we inspected the 200 human-produced APIs to fix any compilation errors and to check whether the participants renamed any parameters. We corrected one compilation error, and we renamed the parameters of 27 human APIs to match the ones automatically generated by APIZATOR. We also removed, from 15 human-produced APIs, variable declarations for return statements that APIZATOR avoids by construction. For example, `int a = b + c; return a;` becomes `return b + c;`.

B. RQ1: Identical APIs

To check for identical APIs, we compared each pair $\langle API_{\mathrm{H}}, API_{\mathrm{A}}\rangle$ with the state-of-the-art source code differencing tool GUMTREE [32]. When comparing the pairs, we excluded differences in method names. GUMTREE implements an Abstract Syntax Tree (AST) differencing algorithm that takes into account fine-grained AST differences while ignoring irrelevant differences in the source code, i.e., new lines, white spaces, and comments.

Fig. 3 shows the distribution of the number of AST differences of the 200 pairs, which ranges from 0 to 99 (average 9.85 and median 7). Interestingly, 63 (31.50 %) APIs generated by APIZATOR are identical to the human-produced ones ($\langle API_{\mathrm{H}}, API_{\mathrm{A}}\rangle$ has zero AST differences). The pair in Fig. 1 is one of such identical APIzations in our experiments.

Achieving identical APIzations is an unrealistic expectation, as in some cases, the participants modified the method body of the API by removing `System.out.println` statements or unnecessary variables. RQ2 and RQ3 give more insights about the dissimilar pairs by studying the APIZATOR effectiveness in extracting the parameters and return statements while ignoring superficial differences in the method bodies.

RQ1 – In summary: APIZATOR generated 63 (31.50 %) APIs identical (including the method-body and import declarations) to the human-produced ones.

C. RQ2: Method Parameters

To answer RQ2, we extracted and compared the parameter lists of the 200 pairs. Given a pair $\langle API_{\mathrm{H}}, API_{\mathrm{A}}\rangle$, we use P_{H} and P_{A} to denote the parameter lists of API_{H} and API_{A}, respectively. Note that the order of elements in the parameter list is irrelevant, thus we considered P_{H} and P_{A} as unordered sets. For example, for the API pair of Fig. 1, $P_{\mathrm{H}} = P_{\mathrm{A}} = \{$`int week`, `int year`$\}$.

Table I breaks down the human-produced APIs (API_{H}) by the number of parameters (the cardinality of P_{H}). The participants produced 58 APIs without parameters, and 142 APIs with one or more parameters (Column "Human APIs " of Table I). The rest of Table I compares P_{H} with the corresponding P_{A}.

Column "$P_{\mathrm{H}} \equiv P_{\mathrm{A}}$" of Table I indicates the number and percentage of APIs pairs with equivalent P_{H} and P_{A}. P_{H} and P_{A} are equivalent if they are both empty, or contain identical parameters. Two parameters $p_h \in P_{\mathrm{H}}$ and $p_a \in P_{\mathrm{A}}$ are identical if and only if they (i) have the same type; (ii) have the same identifier, i.e., variable name; (iii) refer to the same variable in the method body. For example, in the pair of Fig. 1, the parameters `int week` P_{H} and `int week` in P_{A} are identical. They have the same type and identifier, and the two bodies refer to them in the same way. APIZATOR generates 113 (56.50 %) APIs with equivalent parameter lists to the human-produced ones ($P_{\mathrm{H}} \equiv P_{\mathrm{A}}$). When the human-produced APIs have two or more parameters, the number of equivalent pairs decreases. This is an expected result. Intuitively, the more parameters the manually-crafted ground truth API has, the harder it is for APIZATOR to extract an identical parameters list. It is worth mentioning that, in principle, there is no difference if an API has one or more parameters. This is because Algorithm 1 considers each variable in the code snippet individually. In practice, we observed that the majority of human-produced APIzations have at most one parameter. We observed this situation both in the 135 APIs used for extracting the patterns and the 200 APIs used to evaluate APIZATOR (Table I). In fact, the average number of parameters of the 135 APIs is 0.33. The reason for that could be that code snippets often target atomic operations that require one input only.

Column "$|P_{\mathrm{H}} \setminus P_{\mathrm{A}}|$" of Table I shows descriptive statistics (mean, min, median, and max) of the number of missing parameters for each API pair (when $|P_{\mathrm{H}}| \geq 1$). Intuitively, $|P_{\mathrm{H}} \setminus P_{\mathrm{A}}|$ indicates the number of parameters in P_{H} missing from the corresponding P_{A}. The value ranges from 0 to 6 with an average of 0.77 and a median of 0.50. Among the 142 APIs with $|P_{\mathrm{H}}| \geq 1$, 68 of them (47.88 %) have zero missing parameters ($|P_{\mathrm{H}} \setminus P_{\mathrm{A}}| = 0$).

Column "$|P_{\mathrm{H}} \cap P_{\mathrm{A}}|$" of Table I indicates the number of parameters in common between each API pair (when $|P_{\mathrm{H}}| \geq 1$). The value ranges from 0 to 4 with an average of 0.72 and a median of 1.00. Among the 142 APIs with $|P_{\mathrm{H}}| \geq 1$, 91 of them (64.08 %) have at least one parameter in common ($|P_{\mathrm{H}} \cap P_{\mathrm{A}}| \geq 1$). This indicates that APIZATOR often identifies the same parameters that a human would identify.

Column "$|P_{\mathrm{A}} \setminus P_{\mathrm{H}}|$" of Table I shows the number of spurious parameters for each API pair (those extracted by APIZATOR, but not by the human participants). The value ranges from 0 to 5 with an average of 0.23 and a median of 0.00. Among the 200 APIs, 166 of them (83.00 %) do not have spurious parameters ($|P_{\mathrm{A}} \setminus P_{\mathrm{H}}| = 0$). This demonstrates that APIZATOR seldom extracts parameters that a human would not extract.

Column "Jaccard Distance (JD)" of Table I reports the *Jaccard Distance* [33] between P_{H} and P_{A}, and it is defined as $\mathrm{JD}(P_{\mathrm{H}}, P_{\mathrm{A}}) = \frac{|P_{\mathrm{H}} \cap P_{\mathrm{A}}|}{|P_{\mathrm{H}} \cup P_{\mathrm{A}}|}$ from 0 to 1. The lower the value is, the more similar the two sets are. If P_{H} and P_{A} are both empty, $\mathrm{JD}(P_{\mathrm{H}}, P_{\mathrm{A}})$ returns 0.0. The values range from 0.00 to 1.00 with an average of 0.38 and a median of 0.00. These results confirm that in most cases, humans and APIZATOR extracted identical parameter lists. Notably, for nine parameters APIZATOR and the humans extracted the same variables but inferred compatible albeit different types. For example, `java.util.Collection` and `java.util.List`, `double` and `int`. In such cases we consider the parameters to be different.

> **RQ2 – In summary:** APIZATOR generated 113 (56.50 %) APIs with identical parameter lists to the human-produced ones.

TABLE II
RQ3 RETURN STATEMENTS COMPARISON

Return Type				Equivalent Return Type and Statements	
API_{H}	API_{A}	Count	%	Count	%
`void`	`void`	63	31.50	63	100.00
`void`	not `void`	2	1.00	–	–
not `void`	`void`	72	36.00	–	–
not `void`	not `void`	63	31.50	52	82.54
Total		200		115	

D. RQ3: Return Statements

Table II breaks down the 200 APIs pairs by return types (`void` and not `void`). Column "Equivalent Return Type and Statements" counts the number and percentage of APIs with equivalent return statements. A pair of APIs $\langle API_{\mathrm{H}}, API_{\mathrm{A}}\rangle$ has equivalent return statements if (i) both APIs have `void` as return type; or (ii) both APIs return the same type and have identical return statements in the method body. 115 (57.50 %) of the 200 APIs pairs have equivalent return statements. This indicates that APIZATOR can effectively identify the return type and statements that a human would identify.

When both the human and APIZATOR added a return statement (row not `void`, not `void` in Table II), 82.54 % of times they used the same type and return statements. This indicates that the conservative nature of our algorithm leads to few spurious return statements.

> **RQ3 – In summary:** APIZATOR generated 115 (57.50 %) APIs with identical return statements to the human-produced ones.

E. Discussion

Our experimental results are both promising and encouraging. Indeed, for 163 (81.50 %) APIs generated by APIZATOR, either the return statements or method parameters were the same as those produced by the developers. Note that a SO code snippet could have more than one plausible API. Some of the APIs obtained by APIZATOR could be plausible albeit different from the manually-produced ones. Thus, our experimental setup only under-approximates the effectiveness of APIZATOR.

Comparing APIzation efforts. The average APIzation time for the participants ranges from 17 s to 15 min and 58 s, with an average of 4 min and 22 s, and a median of 3 min and 22 s. Note that the participants performed the task offline without our supervision. As such, we cannot tell if a participant was distracted during the experiment. However, these values give an idea of the order of magnitude of the manual effort required. Regarding the 200 code snippets of this experiment, the average execution time of Algorithm 1 was ≈8 s for each code snippet. This shows the potential usefulness of APIZATOR in reducing software development costs. Considering that developers re-use code from SO several times in one day [2], APIZATOR could help speed up the software development process.

False negatives due to literals as parameters. We investigated why some pairs of APIs were different, identifying one main reason (39 cases): *literals-as-parameters*, *when strings and number literals in the arguments of method calls become parameters*.

For example, consider the APIzation in https://apization.netlify.app/evaluation/comparison/8192887/. Both the human and APIZATOR extracted `list` as parameter, but the human also extracted the `String` literal `bea` from `string.matches("(?i)(bea).*")`.

APIZATOR adopts a conservative approach that tolerates missing parameters but minimizes spurious ones, as the results of RQ2 demonstrate. We could have designed APIZATOR to extract all strings and number literals in the method body. Although this would yield fewer false negatives, it would also lead to more spurious parameters since not all string and number literals should become parameters.

We believe that it is better to have false negatives rather than false positives when extracting parameters. This is because extracting literals from the method body "removes" information, which has to be recovered from the SO code snippet. For example, consider the code snippet in Fig. 2 (top). APIZATOR does not extract the string-literal `MD5` as a parameter. Indeed, any random string yields incorrect code. If `MD5` was extracted, the user would need to recover the missing value `MD5` from the SO code snippet. Correctly recognizing and handling the literal-as-parameter issue is an important future work as it will drastically reduce the false negatives of APIZATOR.

Maintainability of the APIs. Currently, APIZATOR returns a dedicated class for each generated APIs. The end users are free to import the class as it is or copy and paste the method and import declarations inside their codebases. Indeed, having many one-method classes results in less cohesive software and ultimately negatively impacts the system's quality. An essential future work would be to propose a technique to group semantically related APIZATOR-generated APIs into the same JAVA class. For instance, one could group APIs that import the same classes and take as input the same type of parameters (e.g., strings, lists, arrays). This will lead to a library of APIZATOR-generated APIs more similar to a manually-written API, facilitating the search, use, and maintainability of APIs automatically extracted from SO.

F. Threats to Validity

Threats to internal validity. A possible threat to internal validity is the choice of the 200 code snippets for the evaluation. They might not be a representative sample of code snippets. We tried to mitigate such a risk by selecting a reasonably large number of snippets for an evaluation involving human participants. Furthermore, by selecting popular code snippets, i.e., based on the views count, we ensured that we selected a relevant sample.

Threats to external validity. A possible threat to external validity is that the four patterns are specific to JAVA, and might not generalize well for other programming languages. For instance, in the case of dynamically-typed languages like PYTHON, the APIzation is easier for some aspects but harder for others. On the one hand, it is difficult to identify possible parameters and return statements by relying on the types of literals. On the other hand, the flexibility of dynamic types allows extracting parameters easier than a statically-typed language like JAVA. Repeating our study for dynamically-typed languages is an important future work.

Another threat to the external validity is that currently APIZATOR only handles two types of compilation errors: missing type declarations and missing variable declarations. APIZATOR cannot produce APIs for those code snippets that have other types of compilation errors. However, these two types are among the most common compilation errors in SO code snippets [5]. APIZATOR relies on previous techniques (CSNIPPEX [5] and BAKER [28]) to fix compilation errors. In the future, APIZATOR could rely on other techniques to handle additional types of compilation error. For instance, a common compilation error in SO code snippets is `compiler.err.expected` [5], which means the code does not comply with the syntax rules of the JAVA language. Examples of such rules are: "a semicolon should be at the end of every statement, or there should be a matching sequence of opening and closing brackets." APIZATOR could rely on a parser that recognizes and fixes such errors.

Threats to construct validity. A possible threat to construct validity relates to the metrics that we used to evaluate APIZATOR. We measured the effectiveness of APIZATOR by counting how many times APIZATOR and the humans made the same APIzation choices. However, a SO code snippet could have more than one plausible API. Additional human evaluators could help recognize when APIZATOR generated a plausible API, albeit different from the human-produced one.

Nevertheless, we preferred to rely on a objective method, even if it might have resulted in a disadvantage for APIZATOR, but is not biased by a subjective evaluation.

VI. Related Work

StackOverflow (SO) provides an important source of crowd-generated data that inspired and powered many techniques and tools. In a recent systematic mapping study, Meldrum et al. identified 266 research papers that rely on SO data to accomplish various software engineering taks [4]. It includes topics like program repair [34], mobile development issues [35], [36], [37], [38], APIs misuses and issues [19], [39], [40], [41], and technology landscape discovery [42], [43]. In this paper, we propose APIZATOR to facilitate the reuse and analysis of SO code snippets by transforming them into compilable and reusable APIs. To the best of our knowledge, it is the first attempt to accomplish this. In the following, we discuss the most related work in code snippet analysis, search, and reuse.

Code snippet analysis. Recently, Terragni et al. proposed CSNIPPEX to resolve compilation errors of SO code snippets [5]. APIZATOR leverages this tool to resolve type declaration errors. Subramanian and Holmes studied the compilability of SO code snippets [6]. However, in the case of missing method declarations, these approaches simply wrap the code snippets in a synthetic method. Differently from APIZATOR, they do not aim at identifying the method parameters and return statements of code snippets.

Researchers have proposed to mine intent-snippet pairs for code summarization or search [44], [45], [46], [7], [47], [24]. The intent of the snippet is often characterized by the SO question title [24]. These techniques analyze the code snippets to identify which lines of code are related to the SO title while filtering out all the implementation details. APIZATOR has the opposite goal of generating the missing implementation details to make the code snippet easy to invoke. All of these techniques aim at identifying the lines of code associated with the intent and do not aim to generate a proper method declaration for the extracted lines of code. APIZATOR could work in synergy with these techniques by creating an API for the code extracted by these techniques.

Code snippet search. There is also a large body of work on improving code search in on-line resources (such as SO) [27], [48], [49]. A popular approach to facilitate search of SO code is to reduce the context switching from IDEs (e.g., INTELLIJ IDEA and ECLIPSE) to web browsers by incorporating SO code search into IDEs. PROMPTER [50] and SEAHAWK [51] recommend SO posts into the IDE based on source code context found in the IDE. T2API [52], NLP2CODE [53], and RACK [54] recommend code snippets extracted from SO based on natural language text describing the programming task. RACK leverages crowd-source knowledge taken from both SO and GITHUB. STACKINTHEFLOW [55] improves the previous approaches by monitoring the behavior of the developers to personalize the retrieved posts. All of these techniques aim to improve the code search or reduce the context switching from IDEs to browsers. Differently from APIZATOR, they do not help developers to integrate the SO code snippet into their code base. APIZATOR complements such approaches, as it could extract, compile and create APIs for the code snippets that are retrieved by these techniques.

Code snippet reuse. Zhang et al. [19] proposed EXAMPLESTACK, a GOOGLE CHROME extension that highlights in a SO page the statements that were changed when a GH developer previously reused the same code snippet. Such highlights help developers to adapt the code snippet in their code bases. To know which statements should be highlighted, EXAMPLESTACK queries an archive of SO code reuses in GH projects. Zhang et al. built such an archive by analyzing 200 code reuses across SO code snippets and GH projects. Similarly to APIZATOR, EXAMPLESTACK aims at facilitating the adaptation of code snippets, but with completely different goals. First, EXAMPLESTACK suggests general code changes [19]. Differently from APIZATOR, it does not automatically extract method parameters and return statements, and it does not aim to generate compilable APIs. Second, EXAMPLESTACK can suggest changes for only those code snippets present in the precomputed archive. Conversely, APIZATOR does not require any prior knowledge on the code snippet under analysis. Third, the input of EXAMPLESTACK and APIZATOR differs substantially. EXAMPLESTACK analyzes parsable code snippets with a well-defined method declaration [19], or by wrapping the snippets with synthetic method headers [6]. Instead, APIZATOR analyzes incomplete code snippets.

VII. Conclusion and Future Work

Online developers forums like StackOverflow (SO) have drastically changed how developers write code [13], [56], [57], [1], [2], [3]. Developers constantly visit SO for finding solutions to programming tasks. The SO revolution has been recognized by the software engineering community and several techniques have been proposed to facilitate the reuse and analysis of SO code snippets [27], [48], [49].

In this paper, we presented APIZATOR, an approach that transforms SO code snippets into compilable and reusable APIs. To the best of our knowledge, this is new to SO code snippet analysis. Our empirical results demonstrate the usefulness of APIZATOR in reducing the developers' effort and enabling the creation of a large dataset of APIs from SO.

There are several possible future works, and we highlight the three most promising ones. First, address the literal-as-parameter issue by employing machine learning to recognize which literal should become a parameter. Second, investigate state-of-the-art approaches [58], [59], [60], [26] to generate semantically meaningful method names. In particular, in our case, one could generate method names by relying on both the natural language free text in the SO posts (e.g., the discussions and comments) and the code snippet itself. Third, explore text summarization and code comment generation approaches [61], [46], [62], [63] to generate the JAVADOC.

REFERENCES

[1] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-Centric Programming: Integrating Web Search into the Development Environment," in *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2010, pp. 513–522.

[2] M.-A. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky, "The (R)Evolution of Social Media in Software Engineering," in *Future of Software Engineering (FoSE)*, 2014.

[3] K. Mao, L. Capra, M. Harman, and Y. Jia, "A Survey of the Use of Crowdsourcing in Software Engineering," *Journal of Systems and Software (JSS)*, vol. 126, pp. 57–84, 2017.

[4] S. Meldrum, S. A. Licorish, and B. T. R. Savarimuthu, "Crowdsourced Knowledge on Stack Overflow: A Systematic Mapping Study," in *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2017, pp. 180–185.

[5] V. Terragni, Y. Liu, and S.-C. Cheung, "CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 118–129.

[6] S. Subramanian and R. Holmes, "Making Sense of Online Code Snippets," in *IEEE Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 85–88.

[7] D. Yang, A. Hussain, and C. V. Lopes, "From Query to Usable Code: An Analysis of Stack Overflow Code Snippets," in *IEEE Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 391–402.

[8] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow," in *IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 25–34.

[9] 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 *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2018, pp. 632–642.

[10] C. Treude and M. P. Robillard, "Understanding Stack Overflow Code Fragments," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 509–513.

[11] R. Holmes and R. J. Walker, "Systematizing Pragmatic Software Reuse," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2012.

[12] V. Terragni and P. Salza. (2021) APIzation: Generating Reusable APIs from StackOverflow Code Snippets - Replication Package. [Online]. Available: https://doi.org/10.5281/zenodo.5236305

[13] B. Vasilescu, A. Serebrenik, P. T. Devanbu, and V. Filkov, "How Social Q&A Sites Are Changing Knowledge Sharing in Open Source Software Communities," in *ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW)*, 2014, pp. 342–354.

[14] L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton, "Improving Low Quality Stack Overflow Post Detection," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 541–544.

[15] M. Gharehyazie, B. Ray, and V. Filkov, "Some from Here, Some from There: Cross-Project Code Reuse in GitHub," in *IEEE Working Conference on Mining Software Repositories (MSR)*, 2017, pp. 291–301.

[16] B. Vasilescu, V. Filkov, and A. Serebrenik, "Stackoverflow and GitHub: Associations Between Software Development and Crowdsourced Knowledge," in *International Conference on Social Computing (SocialCom)*, 2013, pp. 188–195.

[17] A. S. Badashian, A. Esteki, A. Gholipour, A. Hindle, and E. Stroulia, "Involvement, Contribution and Influence in GitHub and Stack Overflow," in *Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2014, pp. 19–33.

[18] S. Baltes, R. Kiefer, and S. Diehl, "Attribution Required: Stack Overflow Code Snippets in GitHub Projects," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2017, pp. 161–163.

[19] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2018, pp. 886–896.

[20] S. Baltes and S. Diehl, "Usage and Attribution of Stack Overflow Code Snippets in GitHub Projects," *Empirical Software Engineering (EMSE)*, vol. 24, no. 3, pp. 1259–1295, 2019.

[21] Google. (2020) Google BigQuery. [Online]. Available: https://cloud.google.com/bigquery

[22] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 7, pp. 654–670, 2002.

[23] J. M. Corbin and A. Strauss, "Grounded Theory Research: Procedures, Canons, and Evaluative Criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[24] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow," in *IEEE Working Conference on Mining Software Repositories (MSR)*, 2018, pp. 476–486.

[25] Explosion. (2020) spaCy. [Online]. Available: https://spacy.io

[26] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S.-W. Lin, "A Neural Model for Method Name Generation from Functional Description," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Feb. 2019, pp. 414–421.

[27] A. Mishne, S. Shoham, and E. Yahav, "Typestate-Based Semantic Code Search Over Partial Programs," in *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012, pp. 997–1016.

[28] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API Documentation," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2014, pp. 643–652.

[29] T. I. Archive. (2019) StackOverflow 2019 Dump. [Online]. Available: https://archive.org/details/stackexchange

[30] F. Rodriguez. (2020) Maven Repository. [Online]. Available: https://mvnrepository.com

[31] C. Treude, O. Barzilay, and M.-A. Storey, "How Do Programmers Ask and Answer Questions on the Web?" in *International Conference on Software Engineering, New Ideas and Emerging Results (ICSE-NIER)*, 2011, pp. 804–807.

[32] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-Grained and Accurate Source Code Differencing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2014, pp. 313–324.

[33] H. Small, "Co-Citation in the Scientific Literature: A New Measure of the Relationship Between Two Documents," *Journal of the American Society for information Science*, vol. 24, no. 4, pp. 264–269, 1973.

[34] X. Liu and H. Zhong, "Mining StackOverflow for Program Repair," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 118–129.

[35] M. L. Vásquez, B. Dit, and D. Poshyvanyk, "An Exploratory Analysis of Mobile Development Issues Using Stack Overflow," in *IEEE Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 93–96.

[36] C. Rosen and E. Shihab, "What Are Mobile Developers Asking About? A Large Scale Study Using Stack Overflow," *Empirical Software Engineering (EMSE)*, vol. 21, no. 3, pp. 1192–1223, 2016.

[37] S. Beyer and M. Pinzger, "A Manual Categorization of Android App Development Issues on Stack Overflow," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 531–535.

[38] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. T. Devanbu, and V. Filkov, "Using and Asking: APIs Used in the Android Market and Asked About in StackOverflow," in *International Conference on Social Informatics (SocInfo)*, 2013, pp. 405–418.

[39] W. Wang, H. Malik, and M. W. Godfrey, "Recommending Posts Concerning API Issues in Developer Q&A Sites," in *IEEE Working Conference on Mining Software Repositories (MSR)*, 2015, pp. 224–234.

[40] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Classifying Stack Overflow Posts on API Issues," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 244–254.

[41] M. L. 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," in *IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2014, pp. 83–94.

[42] C. Chen, Z. Xing, and L. Han, "TechLand: Assisting Technology Landscape Inquiries with Insights from Stack Overflow," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 356–366.

[43] C. Chen and Z. Xing, "Mining Technology Landscape from Stack Overflow," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2016, pp. 14:1–14:10.

[44] Z. Yao, D. S. Weld, W.-P. Chen, and H. Sun, "StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow," in *World Wide Web Conference (WWW)*, 2018, pp. 1693–1703.

[45] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing Source Code Using a Neural Attention Model," in *Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 2073–2083.

[46] E. Wong, J. Yang, and L. Tan, "AutoComment: Mining Question and Answer Sites for Automatic Comment Generation," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 562–567.

[47] A. Zagalsky, O. Barzilay, and A. Yehudai, "Example Overflow: Using Social Media for Code Recommendation," in *International Workshop on Recommendation Systems for Software Engineering (RSSE)*, 2012, pp. 38–42.

[48] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 5, pp. 1069–1087, 2012.

[49] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for Relevant Functions and Their Usages in Millions of Lines of Code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, pp. 37:1–37:30, 2013.

[50] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining Stackoverflow to Turn the IDE into a Self-Confident Programming Prompter," in *IEEE Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 102–111.

[51] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack Overflow in the IDE," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2013, pp. 1295–1298.

[52] T. V. 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 *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2016, pp. 1013–1017.

[53] B. A. Campbell and C. Treude, "NLP2Code: Code Snippet Content Assist Via Natural Language Tasks," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 628–632.

[54] M. M. Rahman, C. K. Roy, and D. Lo, "RACK: Code Search in the IDE Using Crowdsourced Knowledge," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2017, pp. 51–54.

[55] C. Greco, T. Haden, and K. Damevski, "StackInTheFlow: Behavior-Driven Recommendation System for Stack Overflow Posts," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2018, pp. 5–8.

[56] R. Abdalkareem, E. Shihab, and J. Rilling, "What Do Developers Use the Crowd for? A Study Using Stack Overflow," *IEEE Software*, vol. 34, no. 2, pp. 53–60, 2017.

[57] D. Ye, Z. Xing, and N. Kapre, "The Structure and Dynamics of Knowledge Network in Domain-Specific Q&A Sites: A Case Study of Stack Overflow," *Empirical Software Engineering (EMSE)*, vol. 22, no. 1, pp. 375–406, 2017.

[58] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, "Suggesting Natural Method Names to Check Name Consistencies," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2020, pp. 1372–1384.

[59] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to Spot and Refactor Inconsistent Method Names," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019, pp. 1–12.

[60] Z. Gao, X. Xia, J. Grundy, D. Lo, and Y.-F. Li, "Generating Question Titles for Stack Overflow from Mined Code Snippets," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–37, 2020.

[61] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Enriching Documents with Examples: A Corpus Mining Approach," *ACM Transactions on Information Systems (TSE)*, vol. 31, no. 1, pp. 1:1–1:27, Jan. 2013.

[62] C. Treude and M. P. Robillard, "Augmenting API Documentation with Insights from Stack Overflow," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2016, pp. 392–403.

[63] C. Parnin and C. Treude, "Measuring API Documentation on the Web," in *International Workshop on Web 2.0 for Software Engineering (Web2SE)*, 2011.