Automatic Detection and Update Suggestion for Outdated API Names in Documentation

Seonah Lee, *Member, IEEE*, Rongxin Wu, *Member, IEEE*, Shing-Chi Cheung, *Senior Member, IEEE*, and Sungwon Kang, *Member, IEEE*

Abstract—Application programming interfaces (APIs) continually evolve to meet ever-changing user needs, and documentation provides an authoritative reference for their usage. However, API documentation is commonly outdated because nearly all of the associated updates are performed manually. Such outdated documentation, especially with regard to API names, causes major software development issues. In this paper, we propose a method for automatically updating outdated API names in API documentation. Our insight is that API updates in documentation can be derived from API implementation changes between code revisions. To evaluate the proposed method, we applied it to four open source projects. Our evaluation results show that our method, FreshDoc, detects outdated API names in API documentation with 48 percent higher accuracy than the existing state-of-the-art methods do. Moreover, when we checked the updates suggested by FreshDoc against the developers' manual updates in the revised documentation, FreshDoc detected 82 percent of the outdated names. When we reported 40 outdated API names found by FreshDoc via issue tracking systems, developers accepted 75 percent of the suggestions. These evaluation results indicate that FreshDoc can be used as a practical method for the detection and updating of API names in the associated documentation.

Index Terms—Application programming interfaces, documentation, history, software maintenance

1 Introduction

Developers use application programming interfaces (APIs) extensively in software development. API documentation provides an authoritative reference regarding how APIs should be used. Changes regarding API implementations in popular libraries are common. In practice, API implementations change regularly to accommodate user needs [1]. The reasons for such changes include meeting new market demands, enhancing user experience, providing better performance, increasing compatibility and fixing security issues. Developers are advised to build their applications on top of the latest API versions. These API changes are commonly accompanied by updates of manually written documentation, which are typically extensive and tedious. As a result, the APIs in documentation references are frequently outdated [2].

In this paper, we refer to this phenomenon as *the outdated API problem*. Due to this problem, developers can make programming errors and can be confronted with issues such as loss of time and confusion [3], [4]. The outdated API problem would require developers to overhaul their files and correct outdated API names manually every time their system is rebuilt with an updated API library, which is a tedious, time-consuming and error-prone process. If developers can use a tool that automatically detects and updates outdated API names in their documentation, it can significantly reduce the developers' time spent on manually updating their documentation as well as being confused by outdated APIs. Because nowadays software is developed more incrementally and the length of the delivery cycle for the next version of a software system is shortened (*cf.* [5], [6]), the value of such a tool is greater than ever.

In this paper, we address the outdated API problem with the following contributions:

1. We propose an automatic approach for the detection and updating of outdated API names in documentation.
2. We implement our approach as a tool, namely FreshDoc, and evaluate it with real-world software projects. The evaluation results show that our approach significantly outperforms the state-of-the-art approaches, DocRef and AdDoc, in the detection of outdated API names. Moreover, FreshDoc can provide developers with suggestions for updating outdated API names.

To automatically detect and update outdated API names, three challenges must be overcome. First, in manually written documentation, no consistent rules exist to distinguish API names from user-defined terms (e.g., *Spatial4J*). Second, code snippets provide insufficient source information for recovering qualified API names. Third, if the same names reappear in the latest version of the code and its revision histories, it is difficult to determine whether the API names

• *S. Lee is with the Department of Aerospace and Software Engineering and with the Department of Informatics, Gyeongsang National University, Jinju, Jinju-daero 501, Republic of Korea. E-mail: saleese@gnu.ac.kr.*
• *R. Wu and S. C. Cheung are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China. E-mail: {wurongxin, scc}@cse.ust.hk.*
• *S. Kang is with the School of Computing, KAIST373-1, Guseong-dong, Yuseong-gu, Daejeon 34141, Republic of Korea. E-mail: sungwon.kang@kaist.ac.kr.*

Manuscript received 23 Jan. 2018; revised 13 Dec. 2018; accepted 7 Feb. 2019. Date of publication 25 Feb. 2019; date of current version 16 Apr. 2021. (Corresponding author: Sungwon Kang.) Recommended for acceptance by E. Murphy-Hill. Digital Object Identifier no. 10.1109/TSE.2019.2901459

are outdated. The underlying idea of FreshDoc is that an outdated API name can be detected and updated by knowing how the API implementation has been changed across its revisions in the code repository. To overcome these challenges, we developed FreshDoc based on a three-step approach. First, to translate API implementation changes into documentation updates, FreshDoc extracts change rules from the API implementation revisions and then applies the rules. The extracted change rules help distinguish outdated API names from user-defined terms and API names in third-party libraries. Second, to detect outdated API names in API documentation, FreshDoc checks whether the API names are in the revision history but not in the latest version of the library. This step prevents the API names in the latest version from being mistakenly regarded as outdated API names. Third, once FreshDoc has detected outdated API names in the documentation, the program utilizes the change rules for marking and updating outdated API names in the documentation. FreshDoc also uses heuristic rules for recommending new API names. Currently, FreshDoc can be applied to the manually written API documentation of Java projects that maintain their source code revision history in a Git repository.

We evaluated FreshDoc on the manually written documentation of four Java open source projects [7], [8], [9], [10]. The evaluation results show that FreshDoc performs well in the detection of outdated API names, with an F-measure of 60 percent. FreshDoc significantly outperforms the existing state-of-the-art methods DocRef [2] (6 percent F-measure) and AdDoc [12] (12 percent F-measure). Moreover, FreshDoc can update outdated API names with a high degree of accuracy, according to the results of two evaluations. In the first evaluation, we compared the FreshDoc updates with the developers' updates in the revised documentation for the four projects. We found that FreshDoc covered 60 of 73 manual updates. In the second evaluation, we posted the 40 suggestions made by FreshDoc to issue tracking systems. Of the 40 new API outdated problems that were reported, 30 were confirmed and fixed by developers within a few weeks after posting.

The remainder of this paper is organized as follows. Section 2 shows motivating examples. Section 3 addresses challenges. Section 4 describes our method. Section 5 explains our experimental setup, and Section 6 presents the evaluation. Section 7 discusses the usefulness of our method. Section 8 surveys related work, and Section 9 discusses threats to validity. Section 10 concludes the paper.

2 Motivation

This research was inspired by a communication between the first author of this paper and a test engineer in TTA.[1] The test engineer's job was to develop test cases according to a bunch of documentation provided by a company and conduct tests using them. However, when she reported test results, the company people often said that the documentation was out of date. The problem of outdated untrustworthy documentation has also been pointed out in literature: "*Unfortunately, the documentation for most software systems is usually out-of-date and therefore cannot be trusted*" [13], [14].

Outdated documentation can result in poor software maintenance [15], [16], [17], [18], [19], [20], [22]. First, outdated documentation causes software aging [17]. As changes are made to a software system, its documentation becomes increasingly outdated [18]. Due to the inconsistencies between code and documents, both the original developers and the developers who make changes may not understand the software system and the system becomes deteriorated [17]. Second, outdated documentation hinders the effectiveness of documentation [19], [20]. One survey revealed that being up to date is the second important attribute for effective documentation [19]. Another revealed that outdated documentation can block developers from utilizing APIs [20]. Third, outdated documentation causes confusion [3], [21]: "*Developers are usually confused if a method's behavior is different from what is expected on previous Android releases,* [...] *deleting public methods from APIs is a major trigger for questions that are more discussed and of major interest for the community*" [3]. Last, outdated documentation inhibits newcomers from onboarding a software project [22]: "T*he information was outdated* [...]. *So, at least as a newcomer, it was quite challenging to get past those errors*" [22]. As a result, previous empirical studies have called for the identification and marking of outdated information in API documentation[2] [19], [20], [22].

Worse yet, the presence of outdated API names in documentation often results in bugs, causing runtime errors and compilation errors. For example, an application developer who attempted to configure the Logback project by referring to the documentation reported a critical bug regarding the nonexistence of `CyclicBufferTrackerImpl`. This issue report included the *java.lang.ClassNotFound-Exception*, which was a runtime error [24]. Another application developer, who was confused by a code example in the Hibernate-orm documentation, reported a major bug regarding the nonexistence of `Session. getSession`. He found that the `Session` class does not include the `getSession` method. This example was a compilation error [25]. Both examples misled and frustrated the respective developers. Similar bugs are reported in [26], [27], [28] and in StackOverflow [29], [30].

Outdated documentation is a common problem [14], [31], [32], [33]. Tilly et al. stated, "*Regrettably, documentation for most older programs is out-of-date*" [14]. Two surveys revealed that 67∼68 percent of participants stated that the documentation related to their systems is outdated and stated,

1. TTA is Telecommunication Technology Association that tests and certifies IT products. https://www.tta.or.kr/eng/index.jsp

2. To automatically generate API documentation from code comments, the JavaDoc tool can be used. In such a case, when the code is refactored, Eclipse Integrated Development Environments (IDEs) can prefix names with specific annotations (e.g., @link) in comments [11], [19]. However, previous studies have pointed out the useless content of these automatically generated documentations. Maalej and Robillard found that 50% of the content of automatically generated documentation provides no useful information [11]. Forward reported one participant's description: "[automated documentation tools] don't collect the right information" [19]. We have also observed that open-source projects must often manually maintain the written documentation that describes the API usages [7], [8], [9], [10]. Thus, in this paper, we focus on manually written documentation and on how to update the outdated API names within it.

TABLE 1
Statistics for Outdated API Issues in GitHub

Keyword	#Issues
outdated API (*c*/*m*/*f*)	119,081
obsolete API (*c*/*m*/*f*)	125,584
out of date API (*c*/*m*/*f*)	507,971
Total	752,636

c denotes "class," m denotes "method," and f denotes "field."

"*documentation typically suffers from the outdated problems*" [31], [32]. A recent survey reported that 93 percent of participants observed that incomplete or outdated documentation is a pervasive problem [33]. In addition, outdated API issues are pervasive. To identify documentation issues, we searched GitHub for the keyword "outdated API". By also using other keywords such as "obsolete class", we identified 752,636 issues, as shown in Table 1. To understand the details, we surveyed the statistics of all issues, documentation issues and outdated API issues in the issue tracking systems of four projects, as indicated in Table 2. Specifically, to identify documentation issues, we used the keyword "doc", the labels "docs" and "documentation" or the documentation component names of the projects. Table 2 presents the number of issues, the percentage of issues with a priority of "Major" or higher, and the average fixing time of the closed issues for three groups of issues. The numbers of outdated issues range from 45 to 846 in these projects, as shown in the third column from the right in Table 2. We found these issues by using the three representative keywords in Table 1. The percentages of issues classified as major issues were 69 percent for the Logback project and 98 percent for the Hibernate-orm project. We could not calculate the percentages of the major issues for the other two projects because their issue tracking systems do not have priority categories in the issue report. The average fixing time was between 21 and 522 days.

The manual detection and update of outdated APIs in documentation require a significant amount of effort [2], [23]. For example, the DOCUMENT section of Fig. 1 contains outdated APIs. To discover these APIs, developers should examine API names in approximately 30,000 lines of API documentation per project (*cf.* Table 6). Developers who have been working on the project could find outdated APIs with their expertise of the project. However, developers who are unfamiliar with the project may find difficulty to recognize outdated APIs and thus misuse these APIs [34]. The process of updating manually written documentation can be especially exhausting after significant code refactoring. For example, an issue report suggesting a refactoring, "*Refactoring accessors using only getters and setters*" [35], involved 15,078 code revisions in 839 classes. One month later, a contributor left the comment, "*It breaks a lot of examples in the docs, on the mailing list and all over the Internet*." Eight months after the comment was made, another contributor commented on missing documentation updates. The contributor noted the mixes of `hits()` and `getHits()`, as shown in the BUG REPORT section of Fig. 1, leading to subsequent updates [36] of *search.asciidoc*, as shown in the DOCUMENT section. The documentation updates occurred eight months after the relevant API implementation changes were made [37], which are shown in the CODE section.

To ameliorate such a situation, this paper proposes an effective mechanism for automatically detecting outdated APIs and updating them in the associated documentation.

3 Challenges

Our method leverages API implementation changes to detect outdated APIs in manually written documentation.[3] The method addresses three technical challenges, which are discussed in detail in the following sections.

3.1 No Well-Defined Documentation Rules

There are no well-defined rules by which APIs should be described in manually written documentation. Finding outdated APIs in such documentation is significantly more difficult than finding outdated APIs in autogenerated documentation due to the occurrences of many user-defined terms and other third-party APIs. Fig. 2 shows a fragment of documentation for the Elasticsearch project [38]. The project contains multiple user-defined terms that are not APIs. For example, *GeoShape* and *MultiSearch* are titles, and *Spatial4J* is a library name. Therefore, even after removing the terms that match the latest target library's APIs from the documentation [2], the remaining terms are mostly not outdated APIs. In addition, developers may include code snippets that invoke other third-party APIs. For example, the Elasticsearch documentation [7] refers to the `writeValueAsBytes` method, which is an API of another library, *Jackson*. It is challenging to distinguish outdated APIs from user-defined terms and third-party APIs.

3.2 Incomplete Source Information

Insufficient structural information of code snippets in API documentation introduces ambiguity when linking an API to its library. When a code snippet is brief, the qualified name of an API may not be recovered. For example, the `prepareSearch()` method in Fig. 2 has no declared class. Because different classes can have methods and fields with the same names [42], [43], it is difficult to associate the name of a method or a field in the documentation with a specific code in the API library implementation. A search of `prepareSearch()` in the latest version could coincidentally find a method with the same name in other irrelevant classes, leading to the misunderstanding that prepareSearch() is still valid and not outdated. Furthermore, API documentation may contain long-obsolete APIs. For example, Fig. 3 shows a code snippet that contains the `ExecutionContext` class, which was renamed 10 years ago. Outdated APIs can be identified based on the API changes between the releases of source code [12]. However, the outdated APIs that were changed before these releases, but still exist in the documentation, would be missed.

3.3 A Series of Code Changes

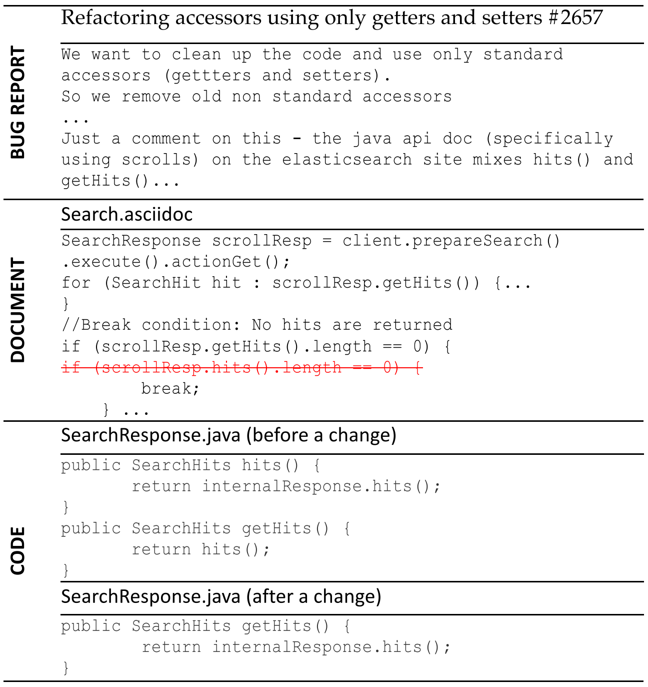
A series of changes can occur to an API through multiple revisions. Because APIs can be deleted and created with the same name at different times and can be renamed multiple

3. We do not consider grammatical/spelling mistakes such as "modes.The" or "follwoing" in documentation because they do not lead to the filing of bug reports, unlike the outdated APIs.

TABLE 2
Statistics for Outdated API Issues

Project	All Issues			Documentation Issues			Outdated API Issues		
	#Issues	Issues ≥ Major	Fixing Time	#Issues	Issues ≥ Major	Fixing Time	#Issues	Issues ≥ Major	Fixing Time
Elasticsearch [38]	7731	N/A	36 days	470	N/A	115 days	159	N/A	35 days
Springboot [39]	2923	N/A	41 days	287	N/A	60 days	71	N/A	21 days
Hibernate-orm [40]	9853	7882 (80%)	336 days	606	448 (74%)	518 days	846	835 (98%)	386 days
Logback [41]	1516	1167 (77%)	268 days	355	209 (59%)	272 days	45	31 (69%)	522 days

times, it is not straightforward to determine the correct API for replacing an outdated API. For example, the `hits()` method in Fig. 1 was renamed to `getHits()`. If the method is renamed to `getSearchHits()`, the method `hits()` should be updated to `getSearchHits()` instead of `getHits()`. Moreover, the polymorphism in a class hierarchy can affect the updating of API names in the documentation. When an API has been deleted, we cannot be sure whether the API has been deleted or has been replaced by its parent. For example, if `hits()` is deleted, one should investigate whether the parent class has an API with the same name. Such complexity arises when determining the correct APIs for API updates.


BUG REPORT
DOCUMENT
CODE

Refactoring accessors using only getters and setters #2657

```
We want to clean up the code and use only standard
accessors (gettters and setters).
So we remove old non standard accessors
...
Just a comment on this - the java api doc (specifically
using scrolls) on the elasticsearch site mixes hits() and
getHits()...
```

Search.asciidoc

```
SearchResponse scrollResp = client.prepareSearch()
.execute().actionGet();
for (SearchHit hit : scrollResp.getHits()) {...
}
//Break condition: No hits are returned
if (scrollResp.getHits().length == 0) {
~~if (scrollResp.hits().length == 0) {~~
        break;
    } ...
```

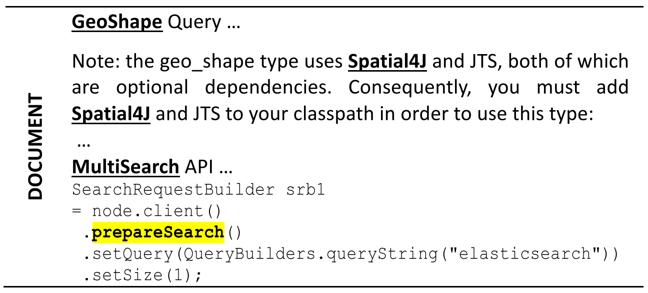
SearchResponse.java (before a change)

```
public SearchHits hits() {
       return internalResponse.hits();
}
public SearchHits getHits() {
       return hits();
}
```

SearchResponse.java (after a change)

```
public SearchHits getHits() {
        return internalResponse.hits();
}
```

Fig. 1. Example of a bug report, outdated documentation and a code change.



GeoShape Query ...

Note: the geo_shape type uses **Spatial4J** and JTS, both of which are optional dependencies. Consequently, you must add **Spatial4J** and JTS to your classpath in order to use this type:

...

MultiSearch API ...

```
SearchRequestBuilder srb1
= node.client()
 .prepareSearch()
 .setQuery(QueryBuilders.queryString("elasticsearch"))
 .setSize(1);
```

Fig. 2. Example of outdated documentation from the Elasticsearch project.

4 The FreshDoc Methodology

We propose a method called FreshDoc. The method automatically updates outdated API names in API documentation. The key idea of FreshDoc is to classify API names as outdated if they appear in the software revision history but not in the latest version of the library. In this section, we first introduce the definition and formulation of the key idea of FreshDoc and then explain the design and the steps of the method.

4.1 Definition and Formulation

An API evolves with the changes in its implementation. To address the API evolution, we first define relevant terms as follows.

Application Programming Interface (API): an interface representing a “contract” between a library and applications with regard to the functions that a library should provide to them.

API implementation: the code that implements an API.

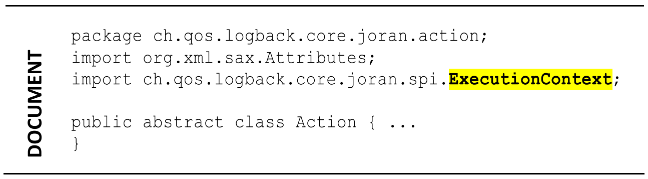
API element: a public class (including an interface), a public method (including its parameters) or a public field in a library.

API signature: the declaration part of an API element (including the parameters of a public method).

API name: a group of terms used to identify an API element (called *name* in this paper).

We then formulate the evolution of an API library implementation as follows. Let a revision history be *RH*, the set of changes in the entire library's revision history be C_{RH} and the set of changes to evolve the library from version $k-1$ to version *k* be $C_{RH}(k)$. A change in $C_{RH}(k)$ can be represented as a pair (e_{k-1}, e_k) of two implementation states of an API element e, where e_{k-1} and e_k are the states just before and just after the change, respectively.

The evolution of API implementation can be used to detect the existence of an outdated API name in documentation. Hence, we define the following relevant terms.



```
package ch.qos.logback.core.joran.action;
import org.xml.sax.Attributes;
import ch.qos.logback.core.joran.spi.ExecutionContext;

public abstract class Action { ...
}
```

Fig. 3. Example of outdated documentation from the Logback project.

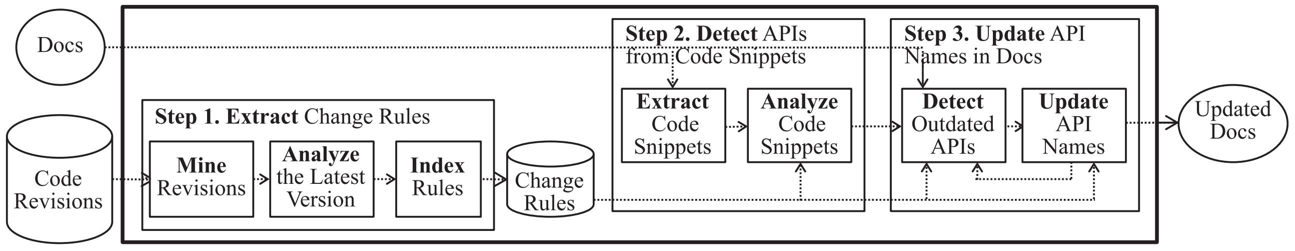


Fig. 4. The steps of FreshDoc.

Documentation: a set of documents (i.e., computer files) that describe the API usages of a library.

Word: a single distinct meaningful element used to form a sentence or a statement in a document.

Qualified name: an unambiguous name with a namespace. In Java, for example, a "qualified name" consists of a sequence of identifiers separated by "." tokens. In the case of a method, the "qualified name" includes the parameters of the method.

Simple name: a nonqualified name without a namespace. A "simple name" consists of a single identifier. In the case of a method, a "simple name" does not include parameters.

In API documentation, an API becomes *outdated* when its documented usage is inconsistent with its current implementation. We formulate a criterion for identifying the outdated API names in documentation as follows. Let the set of API names that are in the revision history be E_{RH} and the set of API names in the latest version be E. Let the name[4] of an API element e in the documentation be $n.e$. If $n.e$ is found in the library's revision history but not in the latest version, then $n.e$ is outdated. This criterion can be expressed as follows:

$$n.e\ is\ outdated\ if\ n.e \in E_{RH} \wedge n.e \notin E_{\omega} \tag{C1}$$

An API name should be updated to avoid inconsistency with the change in its implementation. Let the set of name changes from version $k-1$ to version k be $E_{RH}(k)$. To update an outdated API name, we locate the change $(n.e_{k-1}, n.e_k)$ from E_{RH}. If the change is found, we apply the change $(n.e_{k-1}, n.e_k)$ to $n.e_{k-1}$, which updates $n.e_{k-1}$ to $n.e_k$. The name $n.e_k$ can be updated until it matches the API name in the latest library version, E. This criterion can be expressed as follows:

$$n.e_{k-1} \Rightarrow n.e_k \begin{cases} if\ E_{RH}\ (n.e_{k-1}) = n.e_k\ \wedge\ n.e_k \in\ E_{\omega} \\ Otherwise,\ reapply\ C2\ to\ n.e_k \end{cases}. \tag{C2}$$

So far, we have discussed two criteria for detecting the existence of outdated API names (C1) and for updating the detected inconsistency (C2). However, in the criteria, the name $n.e$ is simplified. The name should be refined to the simple name $s.e$ and the qualified name $q.e$. In the criterion (C1), the first name $n.e$ should be the simple name $s.e$ in the documentation. The second and third occurrences of $n.e$ should be the qualified name $q.e$ in the revision history. The simple name $s.e$ should be recovered to the qualified name $q.e$. In (C2), the name $n.e$ represents the qualified name $q.e$.

To accurately detect outdated API names, we consider three issues associated with the recovery of $s.e$ to $q.e$. First, to utilize structural information of code snippets for the recovery, we must accurately extract the code snippets from the documentation. Second, to avoid mistaking natural language words or user-defined terms for API names, we must distinguish among them. Third, to accurately apply (C1) and (C2), we must associate $s.e$ with its correct $q.e$. Section 4.2 discusses the design strategies implemented in FreshDoc, including the details of the aforementioned issues.

4. Because renaming is the most common refactoring practice [44], we primarily focus on the application of changes to code names.

4.2 Design Strategies for FreshDoc

An overview of FreshDoc is presented in Fig. 4. The first step automatically extracts the change rules from software revision histories (*cf.* Section 4.3). The second step extracts the code snippets from the documentation and analyzes the snippets to recover the qualified names (*cf.* Section 4.4). The third step detects any outdated API names in the documentation and then updates the outdated API names using the change rules (*cf.* Section 4.5).

FreshDoc addresses the challenges described in Section 3 as follows. To overcome the problem of missing well-defined documentation rules (*cf.* Section 3.1), FreshDoc checks whether each word in the documentation matches any API names in E_{RH} as potential API names (*cf.* Section 4.5.1). In more detail, to distinguish API names from natural language words or user-defined terms, FreshDoc utilizes naming convention, programming symbols and change rules (*cf.* Section 4.5.1.1). To detect outdated API names in documentation, FreshDoc checks each word with the context of the word and the extracted change rules. By utilizing the context and the rules, FreshDoc associates a simple name $s.e$ with its qualified name $q.e$ (*cf.* Section 4.5.1.2). As a result, it is possible to exclude the user-defined terms that are not API names as well as to detect the APIs named after legitimate dictionary words, in contrast with DocRef. [2].

To overcome the incomplete source information (*cf.* Section 3.2), FreshDoc extracts code snippets from documentation and analyzes them with an island grammar parser ANTLR [56] (*cf.* Section 4.4). To extract code snippets from documentation, FreshDoc identifies the characteristics of code snippets, text blocks, and the context of a line (*cf.* Section 4.4.1). To recover the qualified names in code snippets, FreshDoc analyzes each code snippet based on the change rules and the structural information extracted from E_{RH} (*cf.* Section 4.4.2). It compensates for insufficient structural information and finally helps to detect the archaic and outdated API names in code snippets. The set of recovered qualified names of code snippets are used to select the most appropriate qualified name (Section 4.5.1).

To update API names induced by a series of code changes (*cf.* Section 3.3), FreshDoc extracts not only change rules [48], [49], [50] but also class hierarchies and structural information (*cf.* Sections 4.3.1 and 4.3.2). In addition, extracting a series of code changes from the entire revision history eliminates the restriction that a user must specify the versions of a library, in contrast to AdDoc [12]. When updating API names, FreshDoc recursively applies the change rules to outdated API names. (Section 4.5.2).

As quality requirements for FreshDoc, we consider accuracy, performance, and simplicity. We make the following decisions:

(1) To yield a reasonable accuracy, we devise techniques of extracting code snippets (Section 4.4.1), detecting outdated API names with change rules (Section 4.5.1) and updating API names (Section 4.5.2).

(2) To yield a reasonable performance, we decide not to extract the entire source code from each revision or use natural language parsers [45], [46]. Instead, we devise a lightweight call analysis technique to analyze only a change set (*cf.* Section 4.3.1.3). The decision enables us to avoid the crashes that occurred when analyzing the entire program of each revision. We also adopt Eclipse Spell Checker to check for spelling errors [47] (Section 4.4.1). The tool lightly analyzes a document but provides sufficient information for the metric DocRef proposed [2].

(3) To simplify the process of updating API names, when $q.e$ appears in several changes in the history *RH*, we retain the latest change of $q.e$ (*cf.* Section 4.3.3). In addition, to select the qualified name from among the possible list of qualified names, Dagenais and Robillard's work [12] performs three different levels of contextual searches. To perform different contextual searches in one shot, we devise a single contextual distance function (*cf.* Section 4.5.1). The function simplifies the selection of the qualified name for w in the context of the documentation.

4.3 Step 1: Extracting Change Rules

To extract the change rules from code revisions, our method combines two previous methods, i.e., Change-Distiller [51], [52] and Call Analysis [48], [49], [50]. Change-Distiller analyzes two versions of a file and extracts the changes to the API definitions [51], [52]. Call Analysis analyzes two versions of a software system and extracts the changes to the API calls [48], [49], [50]. These two methods are complementary to each other in terms of their capability of describing changes. Our method also extends these methods to encompass the changes to an API name.

4.3.1 Mining Revisions

FreshDoc extracts the change rules from all revisions. A change rule, denoted *cr*, represents how an API name is changed to a new name. The rule can be expressed as a 3-tuple, i.e., $cr = (T, L \rightarrow R)$, where T represents the committed revision, L the outdated qualified name, and R the new qualified name. For example, the `hits()` method in Fig. 1 can be expressed as follows:

$$(cc83c2,\ org.elasticsearch.action.search.SearchResponse.hits() \rightarrow org.elasticsearch.action.search.SearchResponse.getHits()).$$

A change rule *cr* can have three different cases:

- $(T, \phi \rightarrow R)$: An API name is created.
- $(T, L \rightarrow \phi)$: An API name is removed.
- $(T, L \rightarrow R)$: An API name is changed to another.

These three cases can be associated with the corresponding actions. For example, when an API element is added, *cr* has the change case $(T, \phi \rightarrow R)$. If the API element is renamed, moved or replaced, *cr* has the change case $(T, L \rightarrow R)$. If the API element is deleted, *cr* has the change case $(T, L \rightarrow \phi)$.

However, when an API element is deprecated, its comment can specify the new API element to replace it with a specific annotation (e.g., `@link`). In this situation, the deprecation action will have the case $(T, L \rightarrow R)$. Subsequently, if the API element is deleted and the previous deprecation action has the case $(T, L \rightarrow R)$, it will be better to preserve *R*. In this situation, the deletion action will retain the case $(T, L \rightarrow R)$.

In addition, alterations to modifiers and annotations affect the use of API names. An API name can be used when it is public and undeprecated: $(T, \phi \rightarrow R)$. An API name is dropped when it becomes private or deprecated: $(T, L \rightarrow \phi)$.

FreshDoc extracts change rules according to the three cases, the nine actions and the three code types (i.e., class, method and field) as specified in Table 3. To extract change rules, FreshDoc uses Git, ChangeDistiller, and Call Analysis in order, with regard to a change set $C_{RH}(k)$. Table 3 presents these cases of change rules, their associated actions, and the extraction methods. We explain the details in the following subsections.

Using Git Features. Using Git [53], FreshDoc detects the addition, deletion, renaming and moving of files in a change set $C_{RH}(k)$. FreshDoc extracts change rules for the classes, methods and fields within the files. To do so, FreshDoc performs the two following steps:

(1) Extract added or deleted classes, methods and fields from an added or deleted file, and create the change rules.

(2) Extract renamed or moved classes, methods and fields from a renamed or moved file using the Git renaming detection function, and create the change rules. Note that renaming and moving files are similar functions. When a file is moved, the qualified name of an API element within the file changes.

4.3.1.2 Adapting ChangeDistiller. FreshDoc uses ChangeDistiller [52] to identify the change patterns of the modified contents of a changed file. Table 3 shows that FreshDoc uses the 18 change patterns of ChangeDistiller to create change rules. For example, the fourth row from the top in the table shows that FreshDoc uses the additional functionality pattern for an added method. The fourth row from the bottom shows that FreshDoc adapts the decreasing accessibility pattern for a deprecated class, method or field.

FreshDoc adapts ChangeDistiller. The adaptation is summarized in the last column of Table 3. First, we modified the decreasing accessibility patterns to detect insertions of deprecated annotations. In this case, we found a new qualified name *R* using doc insert and update patterns (④⑤). Second,

TABLE 3
Extracting Change Rules

Rule	Action	Description	Method	Code Type	Use and Adaptation of ChangeDistiller [53]	
					Change Patterns	Adaptation
$(T, \phi \rightarrow R)$	*Add*	*R* is added.	*G*	*c*	*N/A*	
			CD	*m*	Additional Functionality	
				f	Additional Object State	
	Publicize	*R* becomes public.	*CD*	*c/m/f*	Increasing Accessibility	
	Undeprecate	*R* becomes undeprecated.	*CD*	*c/m/f*	Increasing Accessibility	①
$(T, L \rightarrow R)$	*Rename*	*L* is renamed to *R*.	*G, CD*	*c*	Class Renaming	
				m	Method Renaming, Parameter Insert/Order/TypeChange/ Delete	②
				f	Attribute Renaming	
	Move	*L* is moved to *R*.	*G*	*c/m/f*	*N/A*	
	Replace	*L* is replaced by *R*.	*CD, CA*	*c/m/f*	Statement Insert/Update/Delete, Removed Functionality	③
	Deprecate	*L* becomes deprecated.	*CD*	*m*	Decreasing Accessibility, Doc Insert/Update	④
	Delete	*L* is deleted.	*CD*	*m*	Removed Functionality	
$(T, L \rightarrow \phi)$	*Privatize*	*L* becomes private.	*CD*	*c/m/f*	Decreasing Accessibility	
	Deprecate	*L* becomes deprecated.	*CD*	*c/m/f*	Decreasing Accessibility	⑤
	Delete	*L* is deleted.	*G*	*c*	*N/A*	
			CD	*m*	Removed Functionality	
				f	Removed Object State	

G denotes Git (Section 4.3.1.1), CD denotes ChangeDistiller (Section 4.3.1.2), and CA denotes Call Analysis (Section 4.3.1.4). Code type c denotes a class, m denotes a method, and f denotes a field. The numbers representing adaptions are explained in Section 4.3.1.2.

we modified the increasing accessibility patterns to detect deletions of deprecated annotations (①). Third, we modified the parameter insert, order, type change and delete patterns to identify both the outdated qualified name L and the new qualified name R in the parameter changes (②). Fourth, we combined the removed functionality and statement insert/update patterns to detect replacements (③). The adapted ChangeDistiller can be found in [54]. The adaptation enriches ChangeDistiller to identify code changes, such as detecting the annotation changes concerning API elements.

4.3.1.3 Extracting Structural Information. Performing Lightweight Call Analysis. FreshDoc adopts Call Analysis [48], [49], [50], which captures the replacements of old methods by new methods. Existing methods [48], [49], [50] analyze the entirety of each revision to obtain the qualified names. However, the analysis raises critical performance issues such as crashes when analyzing a significant number of revisions in large software systems, which are the main target of our method. Thus, we developed a lightweight call analysis method that analyzes the change set $C_{RH}(k)$ only and extracts the change rules relevant to the method replacements. The analysis method performs the following steps:

(1) Extract the added and deleted statements from $C_{RH}(k)$.
(2) Check whether each statement contains the simple name of an added member s or a deleted member t in $C_{RH}(k)$.
(3) Track down the methods that contain such statements, and identify these methods as the callers of s or t.
(4) If s has the same set of callers as t, create a replacement rule for s with t, $(T, s \rightarrow t)$.

For example, if there is a deleted method, such as `org.elasticsearch.index.mapper.ParseContext.mappers-Added()`, the analysis checks whether the simple name `mappersAdded()` belongs to each deleted statement (e.g., `context.mapperAdded();`). If so, the analysis adds the method containing the statement (e.g., `org.elasticsearch.index. DocumentMapper.parse()`) as the caller of the deleted method. If the analysis finds an added method (e.g., `org.elasticsearch.index.mapper.ParseContext.mappers-Modified()`) and its simple name `mappersModified()` belongs to the same callers, then it chooses the method as a replacement.

Using a simple name enhances the performance of Call Analysis. However, because simple names are used, our analysis could yield inaccurate call relationships. Therefore, to maintain high accuracy, the analysis uses a threshold θ_α, the minimum number of calls. If the number of calls that change s to t is less than θ_α (i.e., 3), our analysis ignores the replacement. There could also be several candidates of t that replace s. If so, the analysis chooses the one that has the most similar name to s. To avoid extracting inaccurate change rules, the analysis maintains another threshold θ_β. If the similarity value[5] is less than θ_β (i.e., 0.5), our analysis ignores the replacement. Finally, the analysis creates change rules for additional method replacements.

4.3.1.4 Extracting Structural Information. Because the code snippet analysis to be discussed in Section 4.4.2 requires additional structural information, FreshDoc implements a method that extracts the related type of each API element. As shown in Table 4, the related types vary with the code type: the return type for a method and the declared type for a field. For a class, the related type specifies whether it is a class or an interface.

TABLE 4
Structural Information That FreshDoc Identifies

Code Type	API Name	Related Type
Method	qualified name	Return type
Field	qualified name	Declaration type
Class	qualified name	Class | Interface

5. We used the library in https://github.com/rrice/java-string-similarity

Each change rule has its own code type and related type. For example, the change rule of the API name `org.elasticsearch.action.search.SearchResponse.hits()` has its code type `Method` and its related type `SearchHits`.

4.3.2 Analyzing the Latest Version and Its Class Hierarchy

The change rules mined in Section 4.3.1 do not provide sufficient information for the detection of outdated API names because the API names in documentation may be the API names of the latest version of the library. For example, five API elements of the latest version of the Elasticsearch project have the simple name `hits`() in Fig. 1. To prevent the change rules from being mistakenly applied to the up-to-date API names in the documentation, FreshDoc extracts the API information from the latest version. This extraction is accomplished by performing a three-step analysis:

(1) Extract all of the API names from each class *c* of the latest version of the library. Each qualified name $n.e$ is captured by a change rule in the form $(\phi, L \rightarrow \phi)$.
(2) Analyze the structural information of $n.e$ (*cf.* Section 4.3.1.4).
(3) Find *c*'s parent class *pc*. The inheritance relationship is captured as a pair (c, pc) and then stored in a database.

4.3.3 Storing and Indexing Change Rules

FreshDoc stores all of the extracted change rules, denoted *CR*, in a database. Each rule *cr* is also labeled with a tag representing a change action (e.g., *Renamed*). A change rule extracted from the latest version is labeled with a tag "*Latest*". FreshDoc also stores the structural information (*cf.* Section 4.3.1.4) and the class hierarchical information (*cf.* Section 4.3.2). FreshDoc maintains a primary key *PK* by applying the following rules:

$$\begin{array}{ll} L \Rightarrow PK & |\,\text{if } cr\ has\ L \\ R \Rightarrow PK & |\text{if } cr\ has\ no\ L \\ Overwrite\ PK\ | & \text{if } cr\ has\ PK\ that\ exists\ in\ the\ database. \end{array}$$

To improve search performance, FreshDoc creates an index on *CR*. An index holds a simple name with a list of qualified names. FreshDoc identifies the simple name of a primary key *PK* as an index and adds *PK* to the list of qualified names. FreshDoc indexes all change rules, except the change rules labeled *Private*, because they are not API elements.

As an intermediate evaluation, we analyzed 226,166 change rules extracted from the Elasticsearch project [30]. The change rules with the "*Deleted*" tags accounted for 42 percent of all the change rules. The change rules with "*Latest*" accounted for 29 percent, the change rules with "*Renamed*" 17 percent, the change rules with "*Added*" 10 percent, the change rules with "*Replace*" 1 percent, and the others less than 1 percent. This result indicates two points of improvement. First, the change rules with the "*Deleted*" tags account for a higher percentage than we expected. The functionality of many deleted API elements are alternatively provided by other API elements. Second, the change rules with the "*Added*" tags account for approximately 10 percent of all the change rules. Theoretically, these rules should not exist at the end of Step 1 because these rules should have been overwritten by the change rules with the "Latest" tags if they existed in the latest version.

4.4 Step 2: Detecting APIs from Code Snippets

Step 2 consists of the substep of extracting code snippets from documentation (Section 4.4.1) and the substep of recovering the qualified names in code snippets (Section 4.4.2).

4.4.1 Extracting Code Snippets

This section suggests a metric for determining whether a line belongs to a code snippet by checking the characteristics of the line itself and its context. To extract code snippets, DocRef [2] identifies paragraphs by the <p> html tags and checks whether each paragraph is a code snippet by calculating the ratio of natural language errors and counts punctuation marks. However, there are no general marks to identify paragraphs (e.g., <p>) in manually written documentation. Thus, we let FreshDoc identify lines by a line delimiter and check a document *d* line by line.

FreshDoc analyzes the characteristics of code snippets, the characteristics of text blocks and the context of each line. The characteristics of code snippets *cs* and the characteristics of text blocks *tb* are defined as follows.

(1) *cs* is the set of the characteristics of code snippets in a line. The characteristics comprise natural language errors [2], code-related marks (e.g., ";", "(", ")", "$=$", "{","}","$+$" and space dots "."), reserved words (e.g., `abstract`, and `boolean`) and comments (e.g., `//`). To count natural language errors, FreshDoc uses the number of spelling errors found by Eclipse Spell Checker [47].
(2) *tb* is the set of characteristics of text blocks in a line. The characteristics comprise the spaces between words, text-related marks (e.g., "$<<$", "$>>$", and "$==$") and html file expressions (e.g., ".html"). The use of the spaces between words was motivated by our observation that sentences in text descriptions usually connect one word to another by using a space. The text-related marks were markup tags used mainly for titles in AsciiDoc.[6]

By combining all of the characteristics of a line, FreshDoc calculates the following metric:

$$v1 = \frac{2 \times |cs| - |tb|}{|words| + 1} \qquad \text{(E1)}$$

where $|\,words\,|$ is the number of words in a line. In the metric, the number of $|\,cs\,|$ is doubled based on our empirical experiment. If the *v1* value is greater than a threshold θ_χ, FreshDoc regards the line as belonging to a code snippet. For example, in the statement in Fig. 1, "`SearchResponse scrollResp = client.prepareSearch(). execute().actionGet();`", $|\,cs\,|$ is 12, $|\,tb\,|$ is 3, and $|\,words\,|$ is 6. In this case, the value *v1* becomes 3.5. Because the value *v1* is greater than the predefined value of θ_χ (i.e., 0.9), the line is classified as *Code*.

6. AsciiDoc, http://www.methods.co.nz/asciidoc/

Regarding the context of each line, FreshDoc checks whether a line is in a block (e.g., $\{\ldots\}$) and increases $|cs|$ if it is in a block. If a line contains less than two words, FreshDoc regards the line as a continuation of the previous line. If there is a project-specific mark (e.g., —-) to distinguish code snippets from text descriptions, FreshDoc classifies the line as *Tag* and the following line as a different tag from its previous line (e.g., *Text* to *Code*).

It is important to extract a code snippet holistically to analyze its structure. Because our technique uses the characteristics of a line, it could classify code comments as *Text* and split a code snippet into more than one snippet. To prevent this process, FreshDoc classifies a line that has a comment symbol '//' as *Code*.

This extraction technique correctly classified 97 percent (4,243/4,385) of the lines in the documentation of Elasticsearch [7]. Only 142 lines (3 percent) of the text descriptions were misclassified as code snippets. FreshDoc is robust to misclassification of lines because of its two-step analysis. Even if FreshDoc fails to identify API names in the lines misclassified as code snippets in Step 2 (Section 4.4.2), it re-identifies API names in these lines as text in Step 3 (Section 4.5.1).

4.4.2 Analyzing Code Snippets

FreshDoc analyzes and recovers the qualified names of the API elements used in each code snippet. Note that code snippets are usually partial programs and may not be compilable [2], [55]. To address this situation, we leverage an island grammar parser ANTLR [56] to conduct lexical and syntactic analyses of code snippets.

For ANTLR, we define the grammar of code snippets based on the existing Java grammar,7 in which class/interface declarations, method declarations or block statements are considered to be code snippets. We then identify the types of tokens in code snippets. We only need to recover the qualified names of three types of API tokens: class/interface, method and field. For example, in Fig. 1, the method token `hits` and the field token `length` are API tokens, whereas the variable token `scrollResp` and the Java keyword token `break` are not. After identifying the API tokens in the code snippets, FreshDoc recovers their qualified names by using the following three rules:

(R1) Given a class/interface token, FreshDoc selects the old qualified names whose indexes exactly match this token (*cf.* Section 4.3.4). FreshDoc then returns the qualified names that have the code types `Class` (*cf.* Section 4.3.1.4).

(R2) Given a method token m, FreshDoc derives its qualified name in two steps: (a) FreshDoc extracts the class c to which m belongs. If this method invocation is from a variable or a field, FreshDoc binds the class of the variable or the field to m. If this method invocation is from another method $m\prime$, it binds the return type of $m\prime$ to m. Otherwise, we will first find out the classes that contain a method named by m, and then bind these classes to m. (b) FreshDoc selects the old qualified names from the change rules such that the suffix of an old qualified name matches $c.m$. If c has its parent class $c\prime$, FreshDoc finds the qualified names whose suffix matches $c\prime.m$.

(R3) Given a field token, FreshDoc extracts the class/interface to which the field belongs. FreshDoc adopts a strategy similar to R2 to bind the field's type and matches its qualified name.

Using the three rules, FreshDoc shortlists the possible qualified names of API tokens in the code snippets. When applying R2 and R3, FreshDoc typically analyzes each expression from left to right. Given an expression `client.prepareSearch().execute().actionGet()` in Fig. 1, FreshDoc first identifies the class of the `client` instance, then traces the `prepareSearch`() method of the class, and next traces the `execute`() method of the return type of `prepareSearch`().

However, in some expressions, the types of variables or fields may be unknown. In the example, if the class of `client` is unknown, FreshDoc first obtains a candidate set of API names matching `prepareSearch`, leverages the return type of each candidate API element and further infers the remaining part of the expression. If the whole expression of a given candidate API element can be correctly resolved, the qualified name of each API element in the expression can be resolved. Otherwise, the given candidate API element is invalid, and FreshDoc continues to verify the next candidate.

Note that the origination of all outdated API names which are found in an expression of a code snippet should be found in the revision history. In Fig. 1, based on R2, FreshDoc binds `SearchResponse` to the method `hits`(). By checking the change rules, FreshDoc then finds `SearchResponse. hits`() and recovers the qualified name `org.elasticsearch.action.search.SearchResponse.hits`().

4.5 Step 3: Updating API Names in Documentation

Step 3 consists of the substep of detecting outdated API names (Section 4.5.1) and the substep of updating API names (Section 4.5.2).

4.5.1 Detecting Outdated API names

To detect outdated API names in a document, FreshDoc checks each word with the context of the word and the change rules (Section 4.5.1.1). If the word in the document is potentially an API name, FreshDoc recovers the qualified name using the change rules (Section 4.5.1.2). If the qualified name is outdated, FreshDoc regards the API name that corresponds to the word as outdated (Section 4.5.1.3).

4.5.1.1. Identifying a Potential API Name. FreshDoc reads each word w from the document d. FreshDoc then checks whether w is a potential API name. If so, FreshDoc finds the possible qualified names for w, referring to the change rules CR. The steps are as follows:

(1) Check whether w is a potential API name based on naming conventions and programming symbols. If

7. Java Grammar, https://github.com/antlr/grammars-v4/tree/master/java

there is an uppercase in w, '.' immediately before w, or '(' immediately after w, then w is a potential API name. FreshDoc also distinguishes between code types (i.e., class, method and field). For example, because the word "$hits()$" has '(' immediately after "*hits*", FreshDoc regards the code type of $hits()$ as a method.

(2) Find the possible qualified names: (a) If w belongs to any code snippet, FreshDoc obtains qualified names that are recovered from the code snippet (*cf.* Section 4.4.2). This process helps narrow down the possible names by using the structural information contained in the code snippets. (b) If not, or if no names are recovered, then FreshDoc searches for w in the indexes of *CR* and returns the list of qualified names whose indexes match w and code types match the type of w. For example, for $hits()$, FreshDoc returns the list of qualified names, including `org.elasticsearch.action.search.SearchResponse.hits()`.

These steps help address the lack of a set of well-defined documentation rules (*cf.* Section 3.1). Step 1 differentiates API names from natural language words. Step 2 excludes user-defined terms by referring to change rules.

4.5.1.2. Selecting the Qualified Name. FreshDoc selects the qualified name for w in the context of a document. To select the qualified name from a set of candidates, FreshDoc first splits a qualified name into identifiers, creates two vectors for the qualified name and the target document, and calculates the cosine similarity:

$$Similarity = \frac{\vec{A} \bullet \vec{B}}{\left|\vec{A}\right|\left|\vec{B}\right|}, \tag{E2}$$

where vector A represents the qualified name and the second vector B represents the document that contains the identifiers of the qualified name. Each element of A represents each identifier of the qualified name. The length of B is the number of identifiers.

Through the values of two vectors and their similarity metric, FreshDoc implements the following rules:

(R1) The earlier an identifier is in its qualified name, the more general the namespace it represents becomes. The identifier's impact should be reduced.

(R2) If an identifier contains an uppercase letter, it may indicate a class, which should take precedence over its package. The identifier's impact should be amplified.

(R3) The greater the number of identifier instances from the qualified name in a document is, the more likely the qualified name would be selected.

For R1, the number i is assigned to the i_{th} element of A. For R2, if an element represents an identifier with an uppercase letter, its value is doubled. For R3, an element of B is set to $i \times j$ where j is the number of occurrences of the corresponding element (i.e., identifier) in the document.

For example, a qualified name `org.elasticsearch.action.search.SearchResponse.hits()` is divided into six identifiers: *org*, *elasticsearch*, *action*, *search*, *SearchResponse* and $hits()$. Then, vector A becomes [1, 2, 3, 4, 10, 6], where 10 is obtained by doubling 5 by reflecting R2 because the fifth identifier *SearchResponse* contains an uppercase letter. Assuming that a document contains only the sentence "*Please note that the call to hits() is on the SearchResponse API,*" the vector B becomes [0, 0, 0, 0, 10, 6]. Then, the similarity is calculated from $(10^2 + 6^2)/(\sqrt{1^2 + 2^2 + 3^2 + 4^2 + 10^2 + 6^2} \times \sqrt{10^2 + 6^2})$ and the similarity value is 0.91.FreshDoc then measures the distance between w and each identifier that is closest to w and calculates the proximity metric:

$$Proximity = \frac{DS - D}{DS}, \tag{E3}$$

where DS is the length of the document and D is the averaged distance of identifiers in a qualified name.

Through the value of D and the proximity metric, FreshDoc implements the following rules:

(R4) The closer an identifier is to the location of w, the more likely the qualified name would be selected.

(R5) If an identifier is not in the document, the distance is set to the length of the document.

For R4, FreshDoc supports the proximity metric. For R5, FreshDoc sets up the distance of an identifier not in the document d as the length of d. When averaging the distances, FreshDoc reimplements R4 and R5 by reusing vector A. FreshDoc thus calculates a weighted average of these distances D as follows:

$$D = \frac{\sum_{i=1}^{n} |p - q_i| \times a_i}{\sum_{i=1}^{n} a_i}, \tag{E4}$$

where p is the location of w in the document, q_i is the location of each identifier in the qualified name that is nearest to p, n is the number of identifiers in a qualified name, i is the ordinal number of each identifier in the qualified name and a_{i} is the ith element of vector A.

For example, in the sentence "*Please note that the call to hits() is on the SearchResponse API,*" the distance between *hits()* in the sentence and the qualified name above is calculated as follows. First, the nearest location of each identifier from p is found. If p is 7, q_5 for *SearchResponse* is 11, and the distance between p and q_5 is 4. Assuming that a document has only 100 words, the distances are 100 for *org*, *elasticsearch*, *action* and *search*, 4 for *SearchResponse*, and 0 for $hits()$: [100, 100, 100, 100, 4, 0]. Finally, the average distance is calculated, which is 40. In this case, the proximity value becomes 0.6.

Based on the similarity and proximity values, FreshDoc calculates a metric for each qualified name as follows.

$$v2 = similarity \times proximity. \tag{E5}$$

FreshDoc selects a qualified name that has the largest $v2$ value as the most appropriate qualified name for w in the context of the document.

Based on our empirical observations, we made three adjustments. First, in the proximity metric, we amplified the impact of an identifier that has an uppercase letter (R5) by

```
String findName2Update(String q.e_{k-1}) {
   String name-to-update;
   String $q.e_k \leftarrow C_{RH}(q.e_{k-1})$;
   if  ($q.e_k = \phi$) then
      name-to-update ← $\phi$;
      makeSuggestions($q.e_k$);
   else if ($q.e_k \in E_n$) then
      name-to-update ← $q.e_k$;
   else
      name-to-update ← findName2Update($q.e_k$);
   end if
   return name-to-update;
}
```

Fig. 5. Algorithm to find the name to replace an outdated API name.

multiplying its distance by 5. In this case, the distance value for *SearchResponse* in our example becomes 20. We experimented with numbers ranging from 2 to 10 and selected 5 because we observed that when we used a multiplier smaller than 5, FreshDoc misses some outdated API names. Second, we filtered out the qualified names that have a proximity value less than a threshold θ_δ (i.e., 0.6). To find an appropriate threshold, we experimented with $v2$, similarity and proximity metrics. For each metric, we set the initial threshold value to 0.5, increased or decreased the value by 0.1, and checked the experimental results. Finally, we determined the proximity value 0.6 as the threshold θ_δ. Third, we considered rule R6:

(R6) If a qualified name has the tag "Latest", it takes precedence over the other qualified names. The name's impact should be amplified.

We amplified the impact of a qualified name that has a tag of "*Latest*" by doubling the final value $v2$. All of the adjustments were made to the latest version of the documentation for the Elasticsearch project [7].

4.5.1.3 Filtering the Qualified Name. Even if a qualified name is selected for w, it could be a mismatch. The word w could be a legitimate dictionary word (e.g., *Function*, *Percent*, etc.) or a library name (e.g., *JTS*) in the context of the document. Thus, FreshDoc should strictly check the possibility of the selected name to be the name for w. FreshDoc checks whether the package name and the class name are included in the document. If not, FreshDoc filters out the selected name and skips w.

4.5.1.4 Determining the Outdated Name. FreshDoc determines whether the qualified name for w is outdated. If the selected name is outdated, FreshDoc recognizes w as an outdated API name. If the name belongs to the declaration of the API names of other libraries that the target library partially contains, FreshDoc can exclude them by setting up the pattern of a qualifier.

4.5.2 Updating API Names

To update an API name, FreshDoc examines the change rule of the selected name and replaces the API name according to the tags of the change rules, as described in Table 5. For example, if the tag for `org.elasticsearch.action.search.SearchResponse.hits()` is *Replace*, the simple name *hits()* is replaced by the simple name $getHits()$ of `org.elasticsearch.action.search.SearchResponse.getHits()`.

4.5.2.1 Updating API Names by Change Rules. When an outdated API name in documentation has several changes through the revision history, FreshDoc recursively applies the change rules to outdated API names, as indicated in the algorithm shown in Fig. 5. Given an outdated name L (i.e., $q.e_{k-1}$), FreshDoc finds its new name R (i.e., $q.e_k$) by applying the update rules in Table 5 (i.e., $C_{RH}(q.e_{k-1})$).

TABLE 5
Applying Change Rules

Rule	Action	Rule Application
$(T, \phi \rightarrow R)$	*Add, Public, Undeprecate, Latest*	The step skips the update. (If a change rule has the *Latest* tag, the change rule is from the latest version of a target library.)
$(T, L \rightarrow R)$	*Rename, Replace*	The step replaces a simple name of *L* with a simple name of *R*.
	Move	The step finds *L* in d and updates *L* with *R*.
$(T, L \rightarrow \phi)$	*Private*	*N/A* (*cf.*Section 4.3.3)
	Deprecate	The step marks deletion of a simple name of *L*.
	Delete	The step finds the parents recursively according to the class hierarchy: (a) If none of the parents have a member with the same name, it marks *L* as deleted. (b) If one of the parents has a member with the same name, it checks the tag of the change rule of the member and applies the rule.

Once R has been obtained, FreshDoc checks whether the name R belongs to the latest version E_ω. If so, FreshDoc stops the search and returns the name R. If not, the new name R (i.e., $q.e_k$) becomes the name to replace an outdated name L and FreshDoc recursively finds the new qualified name to update L. Whenever R becomes $\boldsymbol{\phi}$, FreshDoc makes suggestions, as described in Section 4.5.2.2. If the simple names of L and R are the same at the end of the search, FreshDoc does not make an update.

FreshDoc generates a report and a document. The generated report is a list of outdated API names with updating information. The information provides the name of the document, the line numbers where the outdated API name exists, the words changed and the change rules applied, as demonstrated by the following example:

$$\ldots/search.asciidoc\ update\ hits\ at\ 70\ from$$
$$org.elasticsearch.action.search.SearchResponse.hits()\ to$$
$$org.elasticsearch.action.search.SearchResponse.getHits().$$

The generated document is an updated API document. When FreshDoc reads a document d, it creates a new document $d\prime$. While reading and checking a word from d, FreshDoc writes the word to $d\prime$. If FreshDoc finds an outdated API name, then it uses the standard html <del> and <ins> tags, as illustrated by the following example:

$$< del > hits < /del >< ins > getHits\ \ < /ins > .$$

The generated report enables users to inspect the findings produced by FreshDoc one by one and update their document. The generated document saves this step when users accept all the findings.

TABLE 6
Details of the Systems

Subject		Source Code (Java)				Revision	Documentation	
Project	Version	#Files	#Lines	#Classes	#Methods	#Commits	#Files	#Lines
Elasticsearch	6.0.0-beta2	5,447	679,327	9,982	64,207	28,718	101	4,385
Spring-boot	2.0.0.M3	3,554	215,649	6,185	22,921	13,266	20	18,042
Hibernate-orm	5.2.11	9,103	662,007	12,552	72,036	8,075	394	39,482
Logback	1.2.3	1,120	57,263	1,263	6,111	3,740	152	53,001
Average		4,806	403,562	7,496	41,319	13,450	167	28,728

4.5.2.2 Suggesting API Names. To update API names, change rules must be utilized. However, more than half of the change rules are found to have no new qualified name. To compensate for it, we developed a suggestion technique. Terragni et al. reported that developers tend to write code snippets using API elements from the same library package or class [57]. Using this and other heuristics, FreshDoc makes suggestions for new API names.

When a new name R is $\boldsymbol{\phi}$, FreshDoc finds new API names to replace the outdated API name, as shown in Fig. 5. To select the candidate API names, the following heuristic rules are applied:

(R1) The candidate API elements belong to the same library package as the outdated API element or its sub packages.
(R2) The API elements have the same code type as the outdated API element.
(R3) The API elements belong to the latest version of the library.
(R4) The names of the API elements are similar to the name of the outdated API element; the names of both share the same word.

To identify each word composing an API name in R4, FreshDoc uses the CamelCase naming convention. FreshDoc splits the API name into one or more words by regarding each uppercase of the API name as the beginning letter of a new word. For example, `AvgBuilder` is split into *Avg* and *Builder*. FreshDoc then checks whether one of the words composing each API name matches that of the outdated API name. For example, because `AvgBuilder` and the outdated API name `Metrics-AggregationBuilder` have the word *Builder* in common, `AvgBuilder` would be the candidate API.

FreshDoc next ranks the candidate API names by counting the number of occurrences of the words composing each API name in the document. The similarity is calculated as follows.

$$Similarity = \frac{\vec{A} \bullet \vec{B}}{\left|\vec{A}\right|\left|\vec{B}\right|}. \tag{E6}$$

where A is a vector whose length is the number of words of the simple API name, and all of its elements are set to 1; B is a vector whose length is the same as that of A, and its elements are the numbers of occurrences of the words of A in the document.

FreshDoc recommends a list of the top five ranked candidate updates in the generated report, as demonstrated by the following example:

$$\begin{aligned}&\ldots\ update\ GeoBoundsBuilder\ to fat\ 14\ from\\ &org.elasticsearch\ldots GeoBoundsBuilder\ to\ \phi\\ &suggestion\ \#1: org.elasticsearch\ldots GeoBoundsAggregationBuilder.\end{aligned}$$

5 Experimental Setup

5.1 Research Questions

To evaluate the effectiveness of FreshDoc, we designed the following five research questions:

RQ1. How accurately does FreshDoc detect outdated API names in the latest version of documentation?
RQ2. How accurately does FreshDoc suggest API name updates in the latest version of documentation?
RQ3. Are the updates made by FreshDoc similar to the human updates found in the revisions of documentation?
RQ4. To what extent do developers accept FreshDoc suggestions?
RQ5. How much time is required to obtain results from FreshDoc?

5.2 Systems for Evaluation

We selected systems for evaluation based on the following criteria. First, such systems should be active, Java-based, open source projects. We selected projects with more than 50 contributors, which implies that the projects are not trivial but interesting to many people. Second, the projects should maintain an issue tracking system so that we can report outdated documentation issues. Third, the projects should maintain manually written documentation as a part of the software revision histories. This requirement allows us to investigate the manual updates of API documentation for RQ3. To select such projects, we first looked into trending projects in GitHub,[8] starting with the top-ranked project. We then checked whether the project has an issue tracking system and maintains its documentation in its revision histories.

As a result, four projects from the GitHub site were selected [38], [39], [40], [41]. Table 6 presents the subject information. The subject section presents the projects and their latest versions. For example, the latest version of the

8. Trending, https://github.com/trending/java?since=monthly

Elasticsearch project was 6.0.0-beta2. The source code section presents the sizes of these projects. On average, the source code contains 4,806 files. The revision section presents the number of commits analyzed from the revision histories. On average, these projects contain 13,450 commits. The documentation section presents the size. On average, the documentation contains 167 files. The documentation is on the GitHub site [7], [8], [9], [10].

We could not find well-defined structures governing the format and contents of the manually written documentation. The documentation of Elasticsearch and Spring-boot is prepared using wiki markups. Hibernate-orm's was prepared in XML and ADOC. Logback's is in HTML. The documentation uses different marks to distinguish among code snippets (e.g., dashes, pre-tags and programlisting tags with CDATA). The contents not only describe the API usages but also explain the configurations. The documentation does not describe API declarations that autogenerated documentation (e.g., JavaDoc) would typically describe.

5.3 Measurements

Based on the differences between API documentation versions and our manual inspection of the latest version, we obtained true positives (tp), false positives (fp) and false negatives (fn) of the number of outdated API locations. We calculated the precision (P), recall (R) and F-measure (F) as follows:

$$P = \frac{tp}{tp + fp} \qquad R = \frac{tp}{tp + fn} \qquad F = \frac{2 \times P \times R}{P + R}.$$

5.4 Procedure

5.4.1 Evaluation of RQ1

RQ1 evaluates the effectiveness of detecting outdated API names in documentation. RQ1 is important because it is a prerequisite for updating outdated API names. To assess the effectiveness of FreshDoc, we compare it with the two state-of-the-art methods DocRef [2] and AdDoc [12] in terms of the detection accuracy. The three techniques proposed different criteria for the detection of outdated API names in their documentation. DocRef detects an API name *e* in the documentation as an error (e.g., an outdated API name), when the API name *e* is neither in the latest version of the library nor declared in its documentation [2]. AdDoc detects an API name *e* in the documentation as a removed API name (i.e., an outdated API name) when *e* is linked to one of the code elements that have been removed between two versions [12]. FreshDoc detects an API name *e* in the documentation as an outdated API name when *e* is not in the latest version but in the software revision history.

We reimplemented DocRef and AdDoc because we are not able to utilize the original tools directly. DocRef [2] was not available online, and the authors also confirmed that the code was not available. AdDoc [12] was available online, but it was outmoded in the given configuration. Thus, we implemented AdDoc's algorithm, especially Algorithm 4 in [12]. In our reimplementation, we focused on the differences between mechanisms, not on minor differences. For example, DocRef uses <pre> or <p> tags to extract code snippets. However, in the manually written documentation, it is difficult to extract code snippets by using these tags. Therefore, we used our improved method of extracting code snippets in the DocRef implementation. The details are irrelevant to the criteria, as described above.

To measure the detection accuracies of DocRef [2], AdDoc [12] and FreshDoc, we applied them to the latest versions of the documentation of the systems in Table 6. For example, we ran our DocRef implementation over the 6.0.0-beta2 version of the Java API documentation of the Elasticsearch project [7] and obtained documentation errors. We then evaluated whether the documentation errors are outdated API names. The detailed procedure used to study RQ1 was as follows:

(1) We ran each tool to detect outdated API names in the documentation of a subject and obtained their reports.
(2) We first investigated the documentation and verified whether the reported API names refer to code elements.
(3) If the reported API names refer to code elements in the documentation, we then searched the code elements in the latest version of the project and verified whether the elements exist.
(4) If the elements were found not to exist, we regarded them as outdated API names. To reconfirm our finding, we manually searched for the elements in Google and examined whether there were discussions on the outdatedness of the API elements. We also checked them in the revision histories.
(5) Because it is infeasible to manually detect outdated API names among numerous API names in the documentation, we collected all of the outdated API elements that were detected by the three tools and confirmed as outdated. We regarded the collection as truths.
(6) Using the grep tool, we searched each API name of the collection in the documentation and expanded the set of truths.

We used the three metrics described in Section 5.3. In the metrics, the denominator of *P* (i.e., $tp + fp$) is the number of outdated API names reported by each tool. The denominator of *R* (i.e., $tp + fn$) is the number of truths. The numerator of *P* and *R* (i.e., *tp*) is the number of outdated APIs reported by each tool and confirmed to be outdated by the set of truths.

As mentioned in (5), manually detecting outdated API names is infeasible. Therefore, as a set of truths, we used the subset of the union of the predictions made by the three tools that we verified to be correct. Our recall value can be used to compare the relative performance of the three tools. In the future, the researchers may be able to choose one of two options to calculate their recall values. First, the researchers can demonstrate the excellence of the proposed method by performing relative comparisons as we have done. Second, based on the outdated APIs found by our method, the researchers can extend the dataset for calculating the recall. We discuss such a usage in Section 7.

5.4.2 Evaluation of RQ2

RQ2 evaluates the effectiveness of updating API names in documentation. This is a novel capability of FreshDoc.

TABLE 7
API Updates Between Two Versions of the Systems

Subject			
Project	Base Ver.	Latest Ver.	#API Updates
Elasticsearch	5.0.0	6.0.0-beta2	31
Spring-boot	1.5.7	2.0.0.M3	30
Hibernate-orm	4.0.0-beta1	5.2.11	11
Logback	1.1.0	1.2.3	1

Because DocRef [2] and AdDoc [12] cannot update API names, we have no comparable tools for RQ2. We performed two evaluation steps to assess RQ2:

First, we calculated how precisely FreshDoc identified change rules for the outdated API names.

(1) We let FreshDoc update the documentation of each subject and obtained the document updates.
(2) We examined the report generated in Section 4.5.2. Because the report included the qualified names recovered from the simple names in documentation, we manually verified whether the outdated API names found in Section 5.4.1 are correctly associated with the appropriate qualified names.
(3) Once the simple name in documentation was recovered to the appropriate qualified name, we determined that we identified a correct change rule.

In the calculation, the denominator of P (i.e., $tp + fp$) is the number of outdated API names identified by FreshDoc and confirmed to be outdated by the set of truths. The numerator of P (i.e., tp) is the number of change rules correctly identified.

Second, we calculated how many API names were correctly replaced with new ones.

(4) We traced the changes of API names in the revision history and checked the changes from the outdated API names to new API names.
(5) We verified whether the new API names exist in the latest version. If the API names exist, we decided that the updates are correct. If not, we decided that the updates are incorrect.

In the calculation, the denominator of P (i.e., $tp + fp$) is the same as the previous denominator. The numerator of P (i.e., tp) is the number of API names correctly replaced with new API names. Because we did not have a set of truths for RQ2, we only calculated the precision metric.

5.4.3 Evaluation of RQ3

RQ3 compares the updates suggested by FreshDoc with the human updates found in the documentation revisions. Because each project includes its documentation folder in the revision history, we can identify developers' manual updates through the revisions of the documentation. We treated the manual updates as the ground truths. Based on these manual updates, we evaluated the effectiveness of FreshDoc. This evaluation supplements the evaluations of RQ1 and RQ2, which only used the latest version of documentation and thus had no ground truths.

For this evaluation, we selected multiple versions of documentation from the base version to the latest version of each project shown in Table 7. For example, we selected the versions from 5.0.0 to 6.0.0-beta2 of the Elasticsearch project. We then adopted the following procedure:

(1) We created a tool to analyze differences between the multiple versions of the documentation. The tool extracts the differences related to code changes by checking whether the changed content includes simple API names.
(2) We reviewed the generated list of changes, manually excluded minor differences (e.g., typos and polishing) and created the list of ground truth updates.
(3) We ran FreshDoc over the base version of API documentation (e.g., the Java API documentation version 5.0.0 of the Elasticsearch project). We obtained the document updates of FreshDoc.
(4) Based on the list of ground truth updates, we evaluated whether FreshDoc correctly updated the outdated API names. The ground truths of the outdated API documentations we used in the evaluation are available online [58].

Table 7 presents the number of manual updates found among documentation versions. For example, we found 31 manual updates between the versions of the Elasticsearch project. We conducted this evaluation for the projects that have API updates as specified in Table 7.

We applied the three metrics in Section 5.3 to this evaluation, conducted based on the past versions. In the metrics, the denominator of P (i.e., $tp + fp$) is the number of outdated API names reported by FreshDoc in past versions. The denominator of R (i.e., $tp + fn$) is the number of ground truths. The numerator of P and R (i.e., tp) is the number of reported API names confirmed as outdated by the set of ground truths.

5.4.4 Evaluation of RQ4

RQ4 collects developers' responses to the FreshDoc suggestions. The developers, who are knowledgeable about the projects, could accept or reject the suggestions. They could also regard the suggestions as major or minor. Through their responses, we determined whether the FreshDoc suggestions are useful to developers in practice. We adopted the following procedure:

(1) To avoid burdening developers, we decided to report the true positives that we identified from the previous evaluations (*cf.* Sections 5.4.1 and 5.4.2).
(2) We regarded the outdated API names appearing in multiple locations in the documentation to be more important than the API names appearing in one location. We thus decided to report the API names appearing in multiple locations.
(3) We posted the FreshDoc suggestions to the issue tracking systems and gathered the developers' feedback.

We repeated this evaluation several times because the documentation and the source code of each project evolved through our research period. We reported our findings in several versions of the projects shown in Table 6.

5.4.5 Evaluation of RQ5

RQ5 asks about the time performance of FreshDoc. We measured the time required to complete each step of FreshDoc

TABLE 8
Evaluation Results of Detecting Outdated APIs in the Latest Version

	All	DocRef						AdDoc						FreshDoc					
Project	#T	#P	#TP	#FP	#FN	P	R	#P	#TP	#FP	#FN	P	R	#P	#TP	#FP	#FN	P	R
Elasticsearch	11	85	4	81	7	0.05	0.36	17	4	13	7	0.24	0.36	7	6	1	5	0.86	0.55
Spring-boot	54	1515	9	1506	45	0.01	0.17	71	10	61	44	0.14	0.19	48	41	7	13	0.85	0.76
Hibernate-orm	95	834	59	775	36	0.07	0.62	4	0	4	95	0.00	0.00	59	42	17	53	0.71	0.44
Logback	37	938	27	911	10	0.03	0.73	4	4	0	33	1.00	0.11	9	8	1	29	0.89	0.22
Total or Average	197	3372	99	3273	98	**0.03**	**0.50**	96	18	78	179	**0.19**	**0.09**	123	97	26	100	**0.79**	**0.49**

#T denotes the number of true positives that all of the three tools found, #P the number of positives that each tool found, #TP the number of true positives, #FP the number of false positives, and #FN the number of false negatives. P denotes Precision and R Recall.

per project using a computer with a 3.7-GHz processor, 32 GB of SDRAM and 512 GB of flash storage.

6 Evaluation Results

6.1 Accuracy of Detecting Outdated APIs

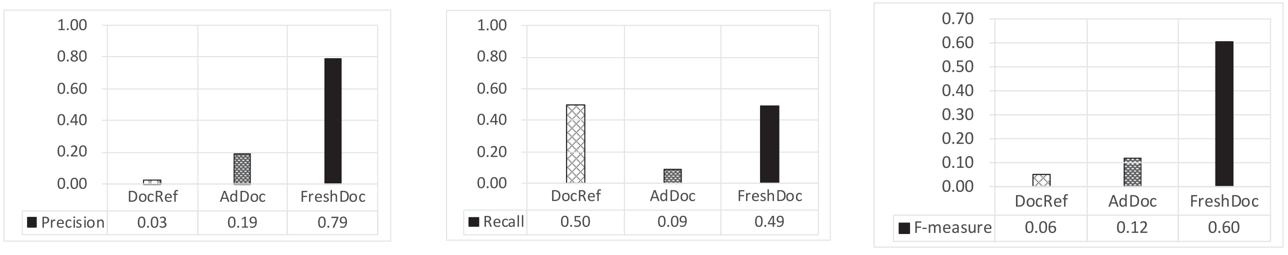
This section reports the accuracies of DocRef, AdDoc and FreshDoc in the detection of outdated API names. Table 8 shows the results. For example, for the Elasticsearch project, 11 true positives (#T) were found by the three tools. DocRef reported 85 outdated API names (#P). Among them, 4 were true positives (#TP). Therefore, the precision of DocRef was 0.05 (P). Because 7 of 11 were not found (#FN), DocRef's recall was 0.36 (R). Likewise, AdDoc reported 17 outdated API names for the Elasticsearch project, 4 of which were determined to be true positives. Therefore, the precision and recall of AdDoc were 0.24 and 0.36, respectively. FreshDoc reported 7 outdated API names, 6 of which were determined to be true positives. The precision and recall of FreshDoc were 0.86 and 0.55, respectively.

In total, DocRef reported 3372 outdated API names, 99 of which were true positives. As a result, DocRef achieved 3 percent precision and 50 percent recall, as shown in the last row of Table 8. DocRef produced many false positives. The tool's detection criterion is to check whether an API name is not in the latest library version and not declared in the documentation. Under that criterion, DocRef incorrectly identifies user-defined terms and other third-party API names as errors. For example, DocRef detects `MongoDB`, `Neo4j`, `Heroku`, `Atomikos`, `Envers`, and `ManyToMany` as errors. In addition, DocRef sometimes misclassifies text descriptions as code snippets and does not properly parse the misclassified descriptions. For that reason, DocRef may miss API names in the text descriptions (e.g., `client. prepareSearch`) and yield false negatives as well. We note that we excluded typos from the DocRef evaluation results.

AdDoc reported 96 outdated API names, 18 of which were true positives. AdDoc achieved 19 precision and 9 percent recall. Interestingly, while AdDoc yielded 100 percent precision for the Logback project, it yielded 0 percent precision for the Hibernate-orm project. The reason is that when detecting outdated API names, AdDoc links API names to the elements that have been deleted between two code releases and does not link API names to the elements in the latest version. For example, `MyEntity` is detected as an error for Hibernate-orm, but the named interface exists in the latest version. AdDoc is prone to false positives when developers maintain many elements with the same names across projects and libraries and repeatedly move classes from one package to another. Moreover, AdDoc produces false negatives because it uses a limited number of releases. We note that in our AdDoc experiment, we used multiple releases, on average 11±2.6. Still, the tool yielded the lowest recall value among the three methods.

FreshDoc reported 123 outdated API names, 97 of which were true positives. FreshDoc achieved 79 precision and 49 ercent recall. Compared with DocRef, FreshDoc demonstrated a significantly higher precision, as shown in Fig. 6a. The main reason for the high precision is the detection criterion of FreshDoc, which checks whether an API name has been declared in the past revision history of the source code but not in the latest library version. Thus, FreshDoc excludes library names, other API names outside the target library and any other user-defined terms. FreshDoc also shows a significantly higher precision than does AdDoc. The reason is that FreshDoc also selects an appropriate qualified name, which exists through the latest version and its revisions, for an API name in documentation.

FreshDoc showed a slightly lower recall than DocRef did but a significantly higher recall than AdDoc did, as shown in Fig. 6b. The main reason for the higher recall resides in the utilization of the entire software revision histories.



(a) Precision (b) Recall (c) F-measure

Fig. 6. Precision, recall and F-measure of detecting outdated API names.

TABLE 9
The FreshDoc Results for Updated APIs

Project	#TP	#D	#R	#S
Elasticsearch	6	3	2	1
Spring-boot	39	31	8	0
Hibernate-orm	42	20	0	22
Logback	8	7	0	1
Total	95	61	10	24

#D denotes the number of deleted APIs, #R the number of updated APIs with new names, and #S the number of APIs that have suggestions.

The static analysis of code snippets and the selection of qualified names based on contextual information also help. When code snippets are extracted accurately, FreshDoc refers to the structural information extracted from revision histories. When the contextual information of API names in the documentation is sufficient to select an appropriate qualified name, FreshDoc selects the qualified name based on contextual information from both code snippets and text descriptions. Thus, although FreshDoc might misclassify text descriptions to code snippets, API names even in the misclassified descriptions can be identified. However, due to contextual selection, FreshDoc misses some outdated API names in documentation.

We characterize when FreshDoc works well. The tool works well when its target project has a long history of code revisions, code snippets are not classified into text descriptions, and the contextual information of API names in documentation is sufficient to select a qualified name. We also characterize when DocRef and AdDoc work better than FreshDoc. Thus, DocRef might work better when its target project has a short history of code revisions, and the documentation has well-defined rules, such as the autogenerated documentation by JavaDoc. In particular, when the target documentation does not contain many user-defined terms, the accuracy of DocRef would be higher. AdDoc might work better when API names are sufficiently unique to select qualified names by analyzing the differences between two releases, and the differences do not contain the cases in which the developers moved code elements from one package to another.

In summary, as shown in Fig. 6c, FreshDoc yielded a higher detection accuracy (60 percent F-measure) than did AdDoc (12 percent) and DocRef (6 percent) when detecting outdated API names.

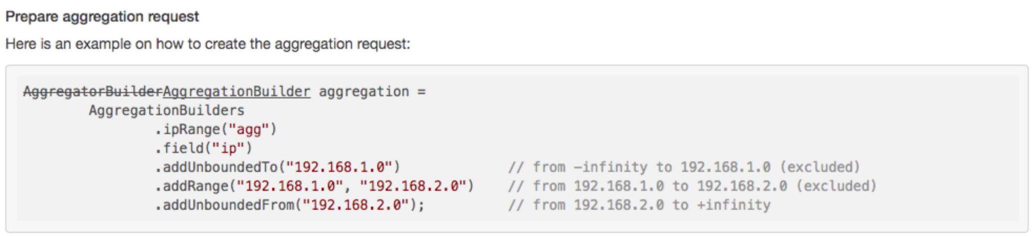
6.2 Accuracy of Updating APIs

After detecting outdated API names in the documentation, the next task is to update the API names. The evaluation on updating the API names consists of two steps, as described in Section 5.4.2. The first step is to evaluate how precisely FreshDoc identified change rules for the outdated API names. Given the true positives of FreshDoc in Section 6.1, we found that once outdated API names have been detected, the matched change rules will be correct. With the correct rules, all of the outdated API names are marked in the documentation. Therefore, the precision for RQ2 is 100 percent. This result guarantees that the outdated API names have not been updated with incorrect API names.

The second step is to evaluate how many of the outdated API names are updated with new API names. To that end, we refined the updates into simply deleted API names that have no new names (#D), the updated API names with new names based on change rules (#R) and the API names that have name suggestions based on heuristic rules (#S). Table 9 summarizes the results. For example, regarding 6 true positives (#TP), the Elasticsearch project contains 3 simply deleted API names (#D), 2 replaced (or renamed) API names (#R) and 1 API name that has suggestions (#S).

Of a total of 95 outdated API names (#TP), 61 API names (#D) were simply deleted (64 percent), 10 API names (#R) were updated with new API names (11 percent), and 24 API names (#S) have suggestions based on heuristic rules (25 percent). Five of the suggestions turned out to be false (5 percent), which means the suggestions for an API name do not include the new API name to correctly replace with. With regard to the replaced API names (#R), FreshDoc updates the outdated API names with new API names by using change rules. With regard to the API names that have suggestions (#S), FreshDoc applies a deletion mark and reports its suggestions. For the simply deleted API names (#D), FreshDoc simply applies a deletion mark. This update can indicate that the detected API names were simply deleted from the code. While this result does not guarantee that the updated text is workable, FreshDoc does inform readers that the updated text is outdated due to API deletion.

Figs. 7 and 8 show examples of the updates made by FreshDoc in documentation. Fig. 7 shows an update in the Elasticsearch document. FreshDoc detects the outdated API name `AggregatorBuilder` in a code snippet and suggests `AggregationBuilder` for the replacement of `AggregatorBuilder`. Fig. 8 shows updates made to the Hibernate-orm document. FreshDoc suggests changing the name `Synchronization` to `InvalidationSynchronization`. The problem we found in these updates is that these documents were written in AsciiDoc. To resolve that issue, we wrapped the used HTML tags with the "+++" mark for AsciiDoc to include raw html. If there was a code block, we placed "`[subs = "quotes"]`" before the block.

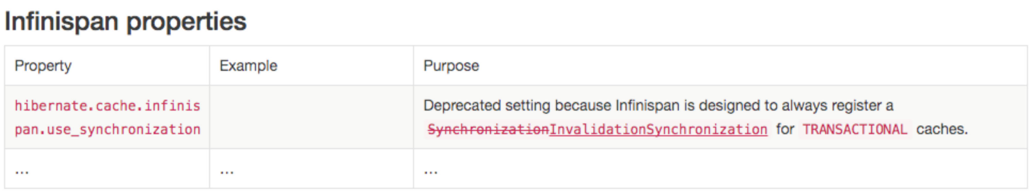


Prepare aggregation request

Here is an example on how to create the aggregation request:

```
~~AggregatorBuilder~~AggregationBuilder aggregation =
        AggregationBuilders
                .ipRange("agg")
                .field("ip")
                .addUnboundedTo("192.168.1.0")             // from -infinity to 192.168.1.0 (excluded)
                .addRange("192.168.1.0", "192.168.2.0")    // from 192.168.1.0 to 192.168.2.0 (excluded)
                .addUnboundedFrom("192.168.2.0");          // from 192.168.2.0 to +infinity
```

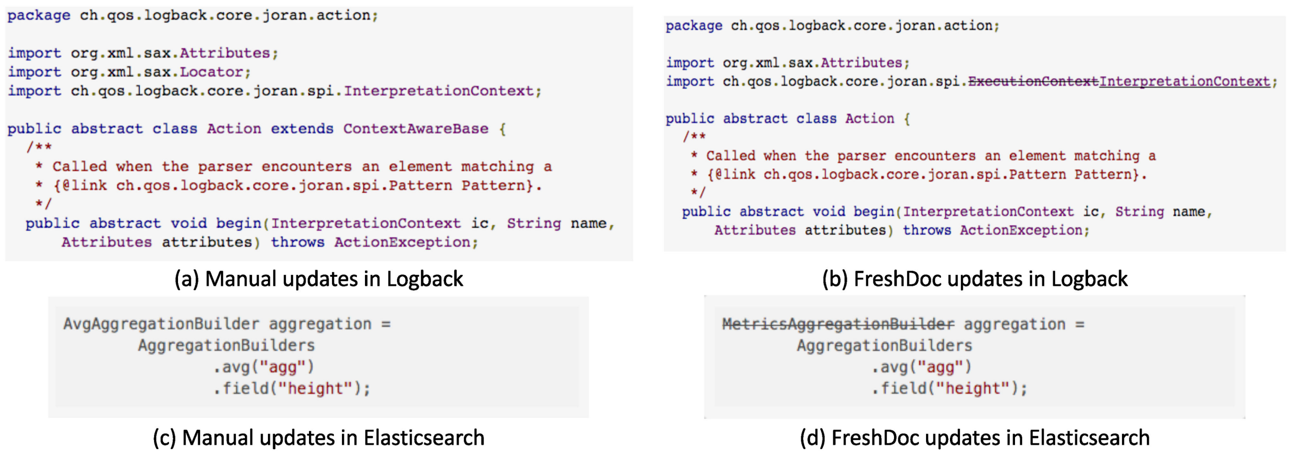
Fig. 7. Code snippet updated by FreshDoc.



Infinispan properties

Property	Example	Purpose
hibernate.cache.infinispan.use_synchronization		Deprecated setting because Infinispan is designed to always register a ~~Synchronization~~InvalidationSynchronization for TRANSACTIONAL caches.
...	...	...

Fig. 8. Text description updated by FreshDoc.



```
package ch.qos.logback.core.joran.action;

import org.xml.sax.Attributes;
import org.xml.sax.Locator;
import ch.qos.logback.core.joran.spi.InterpretationContext;

public abstract class Action extends ContextAwareBase {
  /**
   * Called when the parser encounters an element matching a
   * {@link ch.qos.logback.core.joran.spi.Pattern Pattern}.
   */
  public abstract void begin(InterpretationContext ic, String name,
      Attributes attributes) throws ActionException;
```

(a) Manual updates in Logback

```
package ch.qos.logback.core.joran.action;

import org.xml.sax.Attributes;
import ch.qos.logback.core.joran.spi.~~ExecutionContext~~InterpretationContext;

public abstract class Action {
  /**
   * Called when the parser encounters an element matching a
   * {@link ch.qos.logback.core.joran.spi.Pattern Pattern}.
   */
  public abstract void begin(InterpretationContext ic, String name,
      Attributes attributes) throws ActionException;
```

(b) FreshDoc updates in Logback

```
AvgAggregationBuilder aggregation =
        AggregationBuilders
                .avg("agg")
                .field("height");
```

(c) Manual updates in Elasticsearch

```
~~MetricsAggregationBuilder~~ aggregation =
        AggregationBuilders
                .avg("agg")
                .field("height");
```

(d) FreshDoc updates in Elasticsearch

Fig. 9. Comparison of manual updates and FreshDoc updates.

In summary, FreshDoc correctly updated 31 percent of the detected API names with new API names. The tool correctly updated 11 percent of the detected API names with change rules and suggested new names for 20 percent of the API names. FreshDoc also correctly marked all of the detected API names in the documentation.

6.3 FreshDoc *vs.* Human Updates

Based on the 73 API manual updates in Table 7 of Section 5.4.3, we compared the FreshDoc updates with manual updates. We found that the FreshDoc detected 60 of the 73 manual updates, as shown in Table 10. We also found that 32 of the updates contained the new API names that replaced the outdated ones. Fifteen of them directly contained new APIs in the documentation, as shown in Fig. 9b. Eleven of them were marked *"Deleted"*, as shown in Fig. 9d but had new API suggestions in the generated report.

The comparison revealed the promising result that some of the automatic updates obtained using change rules are similar to human updates. Fig. 9 illustrates this result. The manual update, shown in Fig. 9a, replaces `ExecutionContext` with `InterpretationContext`. The FreshDoc update in Fig. 9b also shows that `ExecutionContext` has been replaced by `Interpretation-Context`. The same result is obtained, but the manual update contains more changes. For example, the manual update additionally imports `org.xml.sax.Locator`, and the `Action` class extends `ContextAwareBase`. A change rule cannot support these changes.

The manual update in Fig. 9c shows the replaced class, `AvgAggregationBuilder`. However, the FreshDoc update in Fig. 9d shows that `MetricsAggregation-Builder` has been deleted. Thus, the mechanism of using change rules fails to find the new API name to replace `MetricsAggregationBuilder` (*cf.* Section 4.5.2.1). In that case, the mechanism of using heuristic rules can work (*cf.* Section 4.5.2.2). Thus, the report generated by FreshDoc suggested `AvgAggregationBuilder` as the top-ranked recommendation to replace `MetricsAggregationBuilder`.

However, our suggestion technique based on the heuristic rules consistently recommends the correct new APIs. The six recommendations of the Hibernate-orm project do not include the correct new APIs.

In summary, FreshDoc detected 60 of the 73 manual updates (82 percent). FreshDoc also updated or suggested 32 of the 53 API names with new API names (60 percent). Of the 32 suggested APIs, 26 were correct (81 percent).

6.4 Developers' Feedback

We have reported 40 outdated API names with the suggestions made by FreshDoc to the issue tracking systems [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70]. Developers accepted 30 of the 40 suggestions (75 percent) and made subsequent updates to the concerned API documentation. Developers responded to our suggestions as follows.

TABLE 10
The FreshDoc Results of Detecting Outdated APIs in the Past Versions and Updating Them

Subject	Detecting Outdated APIs		Updating the Detected APIs		
Project	#MU	#TP	#D	#R	#S
Elasticsearch	31	23	13	3	7
Spring-boot	30	29	14	11	4
Hibernate-orm	11	7	1	0	6
Logback	1	1	0	1	0
Total / Average	73	60	28	15	17

#MU denotes the number of manual updates found in Section 5.4.3. #TP the number of true positives, #D the number of deleted API names, #R the number of updated API names, #S the number of API names that have suggestions.

(1) *Elasticsearch project* [38]: When we posted the question, "*Is the IndexResponse.matches method outdated?*" [59], a contributor replied, "*Thanks. [I] will fix it*," and subsequently fixed it. The contributor left a message saying, "*Docs: IndexResponse.matches () does not exist anymore. Since 1.0, percolator has been redesigned so percolator is not applied anymore at index time.*" When we reported four outdated API names found in version 1.4.4 [60], contributors responded, "*Thanks for reporting this. I'll check that. We might have missed some part of the doc while doing [the code change] #8667*," and then updated the documentation. We also reported five outdated API names found in version 6.0.0 [61]. Contributors responded, "*You are right indeed. The first two are leftovers from #18377; the third item is a leftover from #17650*," and then created two separate tickets for the updates.

(2) *Spring-boot project* [39]: We placed several posts asking whether the FreshDoc suggestions were correct [62], [63], [64], [65]. A contributor responded, "*Well spotted, thanks. The documentation should refer to ConfigurableEmbeddedServletContainer*" [62]. Another contributor responded, "*Thanks. You're right. It was renamed from AutoConfigurationReport to ConditionEvaluationReport*" [63]. Regarding the issue [64] in which we reported three outdated API names found in version 1.2.2, contributors discussed and added the updating tasks to their milestone for the next release. We finally reported four outdated API names found in version 2.0.0.M3 [65]. The contributors commented, "*Looks like we missed these during the big web stack refactor*," and then accepted two of them.

(3) *Hibernate-orm project* [40]: We posted, "*Is jack-event-reg-example outdated?*" in which we reported five outdated API names in the example and suggested new API names [66]. The corresponding contributor replied, "*Looks such as you're correct*." We also reported four outdated APIs in another issue report [68]. The contributors fixed all of them.

(4) *Logback project* [41]: We posted, "*Docs: Are there outdated APIs* (e.g., *SocketRemote) in the site?*", in which we reported four outdated API names including `Execution-Context` of Fig. 3 and suggested new API names [69]. A contributor fixed the outdated API names and asked, "*Finding such stale references seems pretty difficult. How did you do it?*" We also reported two additional outdated API names found in version 1.2.3. Contributors added the issue report to the milestone for version 1.3.0-alpha4 [70].

Regarding the 10 unaccepted API names [64], [65], [67], we found that four were incorrect [64], [65]. Contributors commented that one API name was still used and another was the name of a bean, not an API element [64]. The contributors also replied that one API name was not outdated and another had been relocated to another library [65]. When we reported six outdated API names, the contributors commented that the issue was no longer relevant because a new user guide had been rewritten from scratch [67].

In summary, developers accepted 30 of the 40 suggestions (75 percent) made by FreshDoc and subsequently updated their documentation. Only 4 of the 40 suggestions turned out to be incorrect.

TABLE 11
Execution Time (in hh:mm:ss)

Project	Step 1	Step 2	Step 3	Total
Elasticsearch	2:11:28	0:06:51	0:00:10	2:18:29
Spring-boot	0:21:46	0:00:47	0:00:23	0:22:56
Hibernate-orm	0:51:12	0:02:50	0:01:11	0:55:13
Logback	0:04:39	0:01:28	0:00:36	0:06:43

6.5 Performance

We measured the execution time of FreshDoc. Table 11 presents the execution time of FreshDoc with the systems. FreshDoc took 2 hours 18 minutes and 29 seconds with the Elasticsearch project and less than one hour with the other projects. Because the Elasticsearch project had the greater number of commits, the time needed by FreshDoc appears to be sensitive to the number of revisions.

As indicated in Section 4.2, we decided not to extract the entire source code from each revision and not to use natural language parsers. We observed that these factors negatively contributed to the performance of FreshDoc. For example, when we used existing call analysis approaches (e.g., HiMa [49] and AURA [50]) that analyzed the entire version of a software system, we encountered crashes. To make an objective observation on the overhead, we implemented the checkout of the entire source code from each revision. We found that the simple checkout took 5 hours 24 minutes and 50 seconds. We also made Hunspell, a natural language parser adopted by DocRef, to analyze the documentation of each project. The analysis took 56 minutes and 51 seconds. The alternatives are estimated to take more than twice the time required by FreshDoc.

From our perspective, the execution time is sufficient for the FreshDoc application. Users can obtain the final report and the updated documentation at the end of execution. In addition, to reduce the execution time, we can identify additional approaches. Users can reuse the extracted change rules for updating different documentation and can also allow FreshDoc to incrementally extract change rules whenever a new code change is committed.

7 Discussion

We discuss how FreshDoc can be used and applied. FreshDoc detects, reports and marks outdated API names in documentation. The report generated by FreshDoc reduces library developers' effort in updating the parts of the documentation by indicating where the outdated API names exist in the documentation. The documentation with the outdated API names marked by FreshDoc also helps application developers become aware of outdated API names in documentation and be cautious about the errors caused by the outdated API names. In addition, FreshDoc suggests new API names to replace outdated API names in documentation. Such suggestions reduce developers' efforts in finding new alternative API names.

The use of FreshDoc is not limited to developers' API references and updates in API documentation. For example, FreshDoc can help test engineers as follows. Consider a scenario in which a development team and a testing team are dedicated to a software project, and the development team requests the testing of the source code by passing the code and documentation to the testing team. When a test engineer receives the source code and the documentation describing it, the test engineer can pre-validate the outdated parts of the documentation and request the development team to correct them. Such corrections reduce test engineers' effort by avoiding the development of incorrect test cases with reference to outdated documentation.

FreshDoc can also be applied to the Q&A posts of StackOverflow. We verified that ability by applying FreshDoc to Q&A posts and posting our sample [71]. Marking outdated API names in the posts can help developers referring to these posts recognize the outdated API names in the question and answers and recognize invalid answers. Such recognition would prevent developers from consulting erroneous answers.

In addition, FreshDoc can help researchers collect data related to outdated API names. Once researchers find outdated API names with FreshDoc, it is easy to track down the discussions on the outdated API names. In fact, we found errors caused by outdated API names [24], [25], [26], [27], [28], [29], [30] through a search of the API names detected by FreshDoc. Researchers can collect such data to analyze how many discussions are on outdated API names, what types of errors occur, and how long it takes to fix them. The collection of data related to outdated API names can provide a basis for studying the impact of outdated API elements.

Finally, our research can be extended in at least three new directions. First, our work can be extended to detect and update references to various changes to code (e.g., method parameter changes, exception handling changes, or inheritance relationship changes). Second, our research can be extended to other types of documentation. For example, a design document may not refer to exact API names, but it is likely to refer to similar terms used in creating API names. By tracing the relationships, it would be possible to detect outdated parts of the design document. Third, our work can also be extended to automatically update relevant code and test cases. For example, the method calls in the client code or test cases can be automatically updated according to the evolution of the framework.

8 RELATED WORK

Automatic techniques for API documentation can be broadly divided into three categories: documentation generation [72], [73], [74], [75], documentation error detection [2], [76] and traceability between code and documentation [12], [42], [43], [55].

Techniques in the first category analyze source code and generate documentation [72], [73], [74], [75]. For example, ARENA [74] summarizes code changes and issue reports by analyzing and linking them and generates release notes between two releases. DeltaDoc [72] combines symbolic execution and code summarization techniques to briefly describe code changes and then generates a log message. Other tools automatically generate descriptions of methods or classes [73], [75]. However, these techniques are limited to recovering the missing parts of documentation with no capability of locating and updating outdated APIs. FreshDoc differs from these techniques in that it locates API errors and marks them in a given documentation.

Techniques in the second category detect errors in API documentation [2], [76]. For example, DocRef [2] detects API documentation errors by combining a natural language processing engine with an island parser. DocRef first detects typographical errors in text descriptions and recovers the qualified names of API names in code snippets. DocRef then excludes the APIs of the latest version of a library and the API names declared in the documentation and finally identifies the remaining names as errors. However, DocRef can report many false alarms because its documentation errors still contain many other user-defined terms. Subsequently, Zhou et al. proposed an approach for detecting defects in the directive statements of API documentation [76]. Their method analyzes both code and API documentation and extracts first-order logic constraints from both. If the constraints are inconsistent, the method reports the directive defects. These techniques use a single version of code and documentation as the input. In contrast, FreshDoc analyzes the entire revision history, which helps detect outdated API names accurately. For example, while DocRef fails to filter out many user-defined terms from the documentation errors, FreshDoc can do so because such terms are not API names in the history. FreshDoc also differs from the approach of Zhou et al. approach in that it focuses on detecting and updating outdated API names.

Techniques in the third category recover and utilize the traceability between code and documentation [12], [42] [43], [55]. For example, AdDoc [12] suggests documentation patterns based on the traceability between code and documentation. AdDoc reads two releases of source code and documentation, analyzes documentation patterns of the releases, identifies the differences and suggests documentation additions and deletions. In particular, the deletion algorithm, whose role is similar to that of FreshDoc, links the code terms in the later version of the documentation to code elements in the earlier version of the code. AdDoc then finds the links to the elements that are removed between the earlier and later versions of code and identifies the outdated APIs in the later version of documentation. However, AdDoc mainly focuses on the differences between two specific releases. Thus, AdDoc's applicability is also very restricted because it is based on two unrealistic assumptions, i.e., that the documentation and code have been synchronized in the previous release and that the differences between two releases are minor. In addition, AdDoc is prone to making incorrect links between APIs and code terms, especially when the APIs deleted between the releases have the same names as the APIs in the latest version. The code terms, which must be linked to the APIs in the latest version, are mistakenly linked to the deleted APIs with the same name. Unlike AdDoc, FreshDoc extracts all API names from the entire software revision histories, including the latest version, and selects the most appropriate API name to be linked to a code term in documentation.

In the same category, Baker links StackOverflow posts to API documentation [43]. Baker inputs the posts, constructs a partial abstract syntax tree from code snippets, and identifies the qualified names for the code-like terms by traversing the AST. Baker then pairs the code-like terms in code snippets in StackOverflow with the qualified API names in API documentation. However, Baker is limited to adding the links of code examples to the API documentation, leaving the following question unanswered: What if APIs are deleted or renamed? Moreover, Baker must wait until users upload examples to StackOverflow to add new links of examples. FreshDoc does not require waiting for user-provided examples.

In contrast, our work could be related to call analysis between releases [48], [49], [50] as well as the transformation of code changes [77], [78]. First, to analyze call changes, SemDiff [48] and AURA [50] analyze two releases of a software system. However, these approaches yield inaccurate change rules [48], [49]. HiMa [49] analyzes all of the releases from a revision history. While HiMa yields a higher accuracy, it suffers from performance issues, such as crashes, while analyzing dozens of versions of systems. The reason for these crashes is that HiMa requires the entire code of each release to extract the qualified names of the called APIs. [49]. Our work differs from the reported study in that our call analysis analyzes only the changes between two revisions. Second, to transform code changes, LASE transforms code changes from one code example to another [77]. LASE extracts edit operations (i.e., *insert*, *delete*, *move* and *update*) from an example and applies the operations to another. Likewise, MUSE extracts code examples of a library from its clients, selects representative examples and augments JavaDoc with those examples [78]. FreshDoc differs from these methods in that it mines API implementation changes in revision histories and locates and derives the API updates to synchronize the documentation to the latest APIs of a given library.

Taken together, none of the techniques in these three categories mark outdated API names in documentation and suggest the latest APIs for replacing outdated APIs. Moreover, FreshDoc can detect outdated API names in documentation with a significantly higher accuracy than that achieved by DocRef and AdDoc.

9 Threats to Validity

The manual labelling of the outdated API names as correct or not in the evaluation results may raise threats to internal validity. To mitigate the subjectivity of our manual labelling, we inspected the differences between two API implementation versions and analyzed the associated code and change histories. We also asked project developers for confirmation. Another threat is the versions of DocRef and AdDoc that we used in this work, which may yield a higher or lower detection accuracy than the original versions reported in [2], [12]. To mitigate the differences between the original and the reimplemented versions, we implemented all rules identified in the papers [2], [12]. It should also be noted that the false positives arose not from the implementation details but from the characteristics of DocRef and AdDoc. For example, DocRef identifies user-defined terms as APIs, which leads to false positives. AdDoc identifies the elements removed between two releases as outdated APIs. When the latest release has code elements with the same names as the removed elements, AdDoc yields false positives.

The selection of all target projects from the GitHub site may trigger threats to external validity. We alleviate the threats by selecting large-sized systems with objective selection criteria. Another concern is that systems for evaluation may have been used as part of the FreshDoc development. We clarify that we used the Elasticsearch project for verifying each phase of the FreshDoc approach but did not use the other projects for that purpose; moreover, the differences in accuracy between the four projects were not significant. FreshDoc was applied to the manually written API documentation of a Java project. Although such a project has a different format of documentation from those of the projects used in our evaluation, FreshDoc can still be applied to detect outdated APIs. Nevertheless, some customization could help update outdated APIs in that format and maintain the same level of update accuracy. For example, setting a particular document delimiter can help distinguish code snippets from text descriptions (Section 4.4.1). Additionally, establishing the pattern of a qualifier for a project can narrow down the candidate API names (Section 4.5.1.4). Finally, setting addition and deletion marks can be customized according to the documentation format (Section 4.5.2.1). The customization effort is small in applying FreshDoc to other projects that maintain manually written documentation.

10 Conclusion

This paper describes FreshDoc, a novel method for the detection and updating of outdated API names in documentation. FreshDoc extracts change rules from software revision histories, finds the outdated API names in documentation and updates them with the change rules. We evaluated FreshDoc with the API documentation of four open source projects. Our evaluation results show that FreshDoc automatically identified outdated API names in documentation with 79 precision and 49 percent recall and can correctly update 31 percent of the outdated API names with new API names. In addition, in an evaluation with ground truths, which were found in the documentation revisions, FreshDoc could identify 82 percent of the manual updates and update 35 percent of them with correct new names. In addition, the outdated API names found by FreshDoc were reported as issues, 75 percent of which were subsequently accepted and fixed by developers. The FreshDoc tool and our evaluation results are available at https://figshare.com/articles/FreshDoc Package_zip/7012220 [58] and the FreshDoc code can be found at https://bitbucket.org/docupdater/freshdoc.

In the future, we will improve FreshDoc as follows. First, to increase the accuracy of API detection, we will develop a more systematic mechanism that goes beyond heuristic rules. For example, when extracting code snippets as discussed in Section 4.4.1, we developed a metric based on several heuristic rules. However, if code snippets are classified as text descriptions, there could be a risk that the structural information might be ignored, and the outdated API names in the code snippets might not be detected. We can apply a classification technique to reduce the risk. Second, to improve the precision of API updates, we will expand our set of change rules. For example, as discussed in Section 6.2, 61 of 95 outdated API names were regarded as simply deleted API names and were not updated with new names. We suspect that our set of change rules is not sufficiently comprehensive to update all outdated API names, and we hope to develop a more thorough list of change rules. It would be possible to study more complex change patterns based on developers' behaviors. In addition, one-to-many and many-to-many change rules should be addressed. Based on these change rules, more appropriate lists of updates and recommendations could be revealed. Third, because we focused on API name updates in this paper, we plan to expand FreshDoc to update parameter values. Research on parameter recommendations has already begun but is currently limited to recommending parameters in code [79], [80]. We are studying an automatic technique for updating the parameter values of API calls according to their parameter changes. Fourth, we would also like to apply the updates to the application code. In that case, the updated code should not have any compile or runtime errors, which is a challenge. Finally, we plan to integrate our tool with SCM tools for continuous API updates.

Acknowledgments

This work was supported in part by a grant from National Research Foundation of Korea (NRF-2015R1C1A2A0-1055111, NRF-2018R1D1A1A02085551), and a grant from Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (NRF-2017M3C4A7066210). This work was also supported by Hong Kong RGC/GRF (16202917) and MSRA Collaborative Research Award.

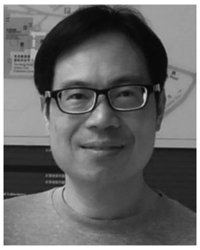
References

[1] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Proc. 29th IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 70–79.
[2] H. Zhong and Z. Su, "Detecting API documentation errors," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 803–816, 2013.
[3] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 83–94.
[4] Elasticsearch, "The book, the documentation and related 1929 community contributions," GitHub, [Online]. Available: https://github.com/elastic/elasticsearch/issues/3789, 2013.
[5] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Reading, MA, USA: Addison-Wesley, 2015.
[6] F. Buschmann, "Industrial-grade DevOps - Balancing agility and speed with extreme quality," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, Art. no. 12.
[7] Elasticsearch, Java api documentation of Elasticsearch project, GitHub, [Online]. Available: https://github.com/elastic/elasticsearch/tree/master/docs/java-api, Accessed Mar. 10, 2019.
[8] Spring-boot, Documentation of Spring-boot project, GitHub [Online]. Available: https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-docs/src/main/asciidoc, Accessed Mar. 10, 2019.
[9] Hibernate-orm, Manual of Hibernate-orm project, GitHub [Online]. Available: https://github.com/hibernate/hibernate-orm/tree/master/documentation/src/main/asciidoc, Accessed Mar. 10, 2019.
[10] Logback, Site of Logback project, GitHub [Online]. Available: https://github.com/qos-ch/logback/tree/master/logback-site, Accessed Mar. 10, 2019.
[11] W. Maalej and M. P. Robillard, "Patterns of knowledge in API reference documentation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1264–1282, Sep. 2013.
[12] B. Dagenais and M. P. Robillard, "Using traceability links to recommend adaptive changes for documentation evolution," *IEEE Trans. Softw. Eng.*, vol. 40, no. 11, pp. 1126–1146, Nov. 2014.
[13] B. Thomas and S. Tilley, "Documentation for software engineers: what is needed to aid system understanding?," in *Proc. 19th Annu. Int. Conf. Comput. Documentation*, 2001, pp. 235–236.
[14] S. R. Tilley, H. A. Müller, and M. A. Orgun, "Documenting software systems with views," in *Proc. 10th Annu. Int. Conf. Syst. Documentation*, 1992, pp. 211–219.
[15] C. Cook and M. Visconti, "Documentation is important," *CrossTalk*, vol. 7, no. 11, 26–30, 1994.
[16] Visconti, Marcello, and Curtis R. Cook, "An overview of industrial software documentation practice," in *Proc. 22nd Int. Conf. Chilean Comput. Sci. Soc.*, 2002, pp. 179–186.
[17] D. L. Parnas, "Software aging," in *Proc. 16th Int. Conf. Softw. Eng.*, 1994, pp. 279–287.
[18] S. R. Tilley, "Documenting-in-the-large vs. documenting-in-the-small," in *Proc. Conf. Centre Adv. Stud. Collaborative Res.: Distrib. Comput.*, 1993, pp. 1083–1090.
[19] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proc. ACM Symp. Document Eng.*, 2002, pp. 26–33.
[20] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Softw.*, vol. 32, no. 4, pp. 68–75, Jul./Aug. 2015.
[21] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Softw. Eng.*, vol. 10, no. 1, pp. 31–55, 2005.
[22] I. Steinmacher, C. Treude, and M. Gerosa, "Let me in: Guidelines for the successful onboarding of newcomers to open source projects," *IEEE Softw.*, (Preprint). 2018, doi: 10.1109/MS.2018.110162131.
[23] B. Dagenais and M. P. Robillard, "Creating and evolving developer documentation: understanding the decisions of open source contributors," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 127–136.
[24] Logback, "CyclicBufferTrackerImpl gone but docs and configuration still refer to," Atlassian [Online]. Available: https://jira.qos.ch/browse/LOGBACK-933, Accessed Mar. 10, 2019.
[25] Hibernate-orm, "Impossible to switch session to EntityMode.MAP," GitHub [Online]. Available: https://hibernate.atlassian.net/browse/HHH-7901, Accessed Mar. 10, 2019.
[26] Elasticsearch, "[DOCS] should use setPostFilter instead of setFilter,"GitHub [Online]. Available: https://github.com/elastic/elasticsearch/pull/5109, Accessed Mar. 10, 2019.
[27] Spring-boot, "Document deprecation of 'spring.view.suffix'," GitHub [Online]. Available: https://github.com/spring-projects/spring-boot/issues/3458, Accessed Mar. 10, 2019.
[28] Spring-boot, "Boot documentation mistake," GitHub [Online]. Available: https://github.com/spring-projects/spring-boot/issues/2871, Accessed Mar. 10, 2019.
[29] Renukeswar Chinta, "java.lang.ClassNotFoundException: org.hibernate. cache.EhCacheProvider," StackOverflow [Online]. Available: https://stackoverflow.com/questions/9643379/java-lang-classnotfoundexception-org-hibernate-cache-ehcacheprovider, Accessed Mar. 10, 2019.
[30] Thepoynt "Hibernate 4 ConnectionProvider Class not found," StackOverflow [Online]. Available: https://stackoverflow.com/questions/23018179/hibernate-4-connectionprovider-class-not-found, Accessed Mar. 10, 2019.
[31] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Softw.*, vol. 20, no. 6, 35–39, Nov./Dec. 2003.
[32] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc. 23rd Annu. Int. Conf. Des. Commun.: Documenting Des. Pervasive Inf.*, 2005, pp. 68–75.
[33] F. Zlotnick, The Open Source Survey, GitHub [Online]. Available: https://opensourcesurvey.org/2017. Accessed Mar. 10, 2019.
[34] M. Wen, Y. Liu, R. Wu, X. Xie, S.C. Cheung, and Z. Su, "Exposing library API misuses via mutation analysis," in *Proc. Int. Conf. Softw. Eng.*, 2019. (Accepted).
[35] Elasticsearch, "Refactoring accessors using only getters and setters," GitHub [Online]. Available: https://github.com/elastic/elasticsearch/issues/2657, Accessed Mar. 10, 2019.
[36] Elasticsearch, "Changed documentation to use getter notation," GitHub [Online]. Available: https://github.com/elastic/elasticsearch/commit/f0cf97c0ac31c309bcc39f54eb90d62c01a7b21b, Accessed Mar. 10, 2019.
[37] Elasticsearch, "refactoring getter/setters for package index.get #2657", GitHub [Online]. Available: https://github.com/elastic/elasticsearch/commit/9d8f503f1da45059bb5cdc31df4eece485869710, Accessed Mar. 10, 2019.

[38] Elasticsearch, Elasticsearch: A Distributed RESTful Search Engine project, GitHub [Online]. Available: https://github.com/elastic/elasticsearch, Accessed Mar. 10, 2019.
[39] Hibernate-orm, Hibernate-orm project, GitHub [Online]. Available: https://github.com/hibernate/hibernate-orm, Accessed Mar. 10, 2019.
[40] Hibernate-orm, Hibernate-orm project, GitHub [Online]. Available: https://github.com/hibernate/hibernate-orm, Accessed Mar. 10, 2019.
[41] Logback, Logback project, GitHub [Online]. Available: https://github.com/qos-ch/logback, Accessed Mar. 10, 2019.
[42] B. Dagenais and M. P. Robillard, "Recovering traceability links between an API and its learning resources," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 47–57.
[43] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 643–652.
[44] E. Murphy-Hill, C. Parnin, and P. B. Andrew, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan./Feb. 2012.
[45] Apache OpenNLP, The Apache Software Foundation [Online]. Available: https://opennlp.apache.org/, Accessed Mar. 10, 2019.
[46] Language Tool, LGPL [Online]. Available: https://languagetool.org/, Accessed Mar. 10, 2019.
[47] Eclipse Spell Checker, GitHub [Online]. Available: https://github.com/eclipse/eclipse.jdt.ui/blob/master/org.eclipse.jdt.ui/ui/org/eclipse/jdt/internal/ui/text/spelling/engine/ISpellCheckEngine.java, Accessed Mar. 10, 2019.
[48] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Trans. Softw. Eng. Methodology*, vol. 20, no. 4, 2011, Art. no. 19.
[49] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 353–363.
[50] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: A hybrid approach to identify framework evolution," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 325–334.
[51] B. Fluri and H. Gall, "Classifying change types for qualifying change couplings," in *Proc. 14th IEEE Int. Conf. Program Comprehension*, 2006, pp. 35–45.
[52] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, 2007.
[53] JGit, Eclipse Foundation [Online]. Available: [Online]. Available: https://eclipse.org/jgit/, Accessed Mar. 10, 2019.
[54] ChangeDistiller (Adapted), [Online]. Available: https://bitbucket.org/docupdater/changedistiller/, Accessed Mar. 10, 2019.
[55] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 832–841.
[56] T. Parr, ANTLR, ANTLR / Terence Parr [Online]. Available: http://www.antlr.org/, Accessed Mar. 10, 2019.
[57] V. Terragni, Y. Liu, and S. C. Cheung, "CSNIPPEX: Automated synthesis of compilable code snippets from Q&A sites," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 118–129.
[58] Seonah Lee "FreshDoc, Tool and Experimental results," FigShare [Online]. Available: https://figshare.com/articles/FreshDoc_Package_zip/7012220, Accessed Mar. 10, 2019.
[59] Elasticsearch, "Docs: Is the IndexResponse.matches()method outdated?" GitHub [Online]. Available: https://github.com/elastic/elasticsearch/issues/7548, Accessed Mar. 10, 2019.
[60] Elasticsearch, "Docs: Are there outdated APIs (e.g., Date-Histogram) in Java API docs?" GitHub [Online]. Available: https://github.com/elastic/elasticsearch/issues/9976, Accessed Mar. 10, 2019.
[61] Elasticsearch, "Docs: Are there outdated APIs (e.g., Aggregator-Builder) in Java API docs?" GitHub [Online]. Available: https://github.com/elastic/elasticsearch/issues/28114, Accessed Mar. 10, 2019.
[62] Spring-boot, "Docs: Is ConfigurableEmbeddedServletCon-tainer-Factory outdated?" GitHub [Online]. Available: https://github.com/spring-projects/spring-boot/issues/1500, Accessed Mar. 10, 2019.
[63] Spring-boot, "Docs: Is AutoConfigurationReport out-dated?" GitHub [Online]. Available: https://github.com/spring-projects/spring-boot/issues/2493, Accessed Mar. 10, 2019.
[64] Spring-boot, "Docs: Are there outdated APIs in the docs?" GitHub [Online]. Available: https://github.com/spring-projects/spring-boot/issues/2598, Accessed Mar. 10, 2019
[65] Spring-boot, "Docs: Are there outdated APIs (e.g., Servlet-Web-ServerFactoryCustomizer) in the docs?" GitHub [Online]. Available: https://github.com/spring-projects/spring-boot/issues/11529, Accessed Mar. 10, 2019.
[66] Hibernate-orm, "Docs: Is jack-event-reg-example outdated?", Atlassian [Online]. Available: https://hibernate.atlassian.net/browse/HHH-9616, Accessed Mar. 10, 2019.
[67] Hibernate-orm, Docs: Are there outdated APIs in the Man-ual (EN-US)?, Atlassian [Online]. Available: https://hibernate.atlassian.net/browse/HHH-9650, Accessed Mar. 10, 2019.
[68] Hibernate-orm, "Docs: Are there outdated APIs (e.g., Infinispan-RegionFactory) in the User Guide?" Atlassian [Online]. Available: https://hibernate.atlassian.net/browse/HHH-12200, Accessed Mar. 10, 2019.
[69] Logback, "Docs: Are there outdated APIs (e.g., SocketRemote) in the site?" Atlassian [Online]. Available: https://jira.qos.ch/browse/LOGBACK-1053, Accessed Mar. 10, 2019.
[70] Logback, "Docs: Are there outdated APIs (e.g., WriterAp- 2112 pender) in the site?", Atlassian [Online]. Available: https://jira.qos.ch/browse/LOGBACK-1367, Accessed Mar. 10, 2019.
[71] Salee, Is ImmutableSettings removed in Elasticsearch? StackOverflow [Online]. Available: https://stackoverflow.com/questions/33115504/is-immutablesettings-removed-in-Elasticsearch, Accessed Mar. 10, 2019.
[72] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proc. IEEE/ACM Int. Conf. Autom. Soft. Eng.*, 2010, pp. 33–42.
[73] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 279–290.
[74] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 484–495.
[75] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2010, pp. 43–52.
[76] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *Proc. 39th Int. Conf. Softw. Eng.*, pp. 27–37, 2017.
[77] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 502–511.
[78] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can I use this method?," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, pp. 880–890, 2015.
[79] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 826–836.
[80] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider, "Exploring API method parameter recommendations," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 271–280.

Seonah Lee received the BS and MS degrees in computer science and engineering from Ewha Womans University, in 1997 and 1999, respectively, she received the MSE degree from the School of Computer Science, Carnegie Mellon University, in 2005, and the PhD degree from the School of Computer Science, KAIST, in 2013. She worked as a software engineer with Samsung Electronics from 1999 to 2006. She served as a research professor at KAIST. Currently, she is an assistant professor of aerospace and software engineering at Gyeongsang National University. Her research interests include software evolution, documentation updates, requirement traceability, software architecture, and data mining. She is a member of the IEEE.

Rongxin Wu received the PhD degree from HKUST, in 2017. He is a post-doctoral research fellow in the department of computer science and engineering at the Hong Kong University of Science and Technology (HKUST). His research interests include program analysis, software security, and mining software repository. His research work has been regularly published in top conferences and journals in the research communities of program languages and software engineering, including POPL, PLDI, ICSE, FSE, ISSTA, *Association for Science Education*, *IEEE Transactions on Software Engineering* and *Empirical Software Engineering* and so on. He has served as a reviewer in reputable international journals and a program committee member in several international conferences. He has ever received ACM SIGSOFT Distinguished Paper award. More information about him can be found at: http://home.cse.ust.hk/∼wurongxin/. He is a member of the IEEE.



Shing-Chi Cheung received the doctoral degree in computing from the Imperial College London. He joined the Hong Kong University of Science and Technology (HKUST) where he was a professor of Computer Science and Engineering, in 1994. He founded the CASTLE research group at HKUST and co-founded, in 2006 the International Workshop on Automation of Software Testing (AST). He was the general chair of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). He was an editorial board member of the *IEEE Transactions on Software Engineering* (TSE, 2006-9). His research interests focus on the quality enhancement of software for mobile, web, deep learning, open-source and end-user applications. He is an ACM distinguished scientist. More information about his CASTLE research group can be found at http://sccpu2.cse.ust.hk/castle/people.html. He is a senior member of the IEEE.

Sungwon Kang received BA degree from Seoul National University, in 1982 and the MS and PhD degree in computer science from the University of Iowa, in 1989 and 1992, respectively. From 1993, he was a principal researcher of Korea Telecom R & D Group until October 2001 when he joined the School of Computing at Korea Advanced Institute of Science and Technology. During 2003-2014, he was an adjunct faculty of Carnegie-Mellon University for the Master of Software Engineering Program. He served as chairs and program chairs of numerous international conferences, the editor of the *Korean Journal of Software Engineering Society*, and as the president of the Korean Software Engineering Society. His research areas include software architecture, software product line, software testing and data-based software engineering.