Noname manuscript No.
(will be inserted by the editor)

How do Programmers Fix Bugs as Workarounds?

An Empirical Study on Apache Projects

Aoyang Yan · Hao Zhong · Daohan Song · Li Jia

Received: date / Accepted: date

Abstract In software development, issue tracker systems are widely used to manage bug reports. In such a system, a bug report can be filed, diagnosed, assigned, and fixed. In the standard process, a bug can be resolved as *fixed*, *invalid*, *duplicated* or *won't fix*. Although the above resolutions are well-defined and easy to understand, a bug report can end with a less known resolution, *i.e.*, *workaround*. Compared with other resolutions, the definition of workarounds is more ambiguous. Besides the problem that is reported in a bug report, the resolution of a workaround raises more questions. Some questions are important for users, especially those programmers who build their projects upon others (*e.g.*, libraries). Although some early studies have been conducted to analyze API workarounds, many research questions on workarounds are still open. For example, which bugs are resolved as workarounds? Why is a bug report resolved as workarounds? What are the repairs and impacts of workarounds? In this paper, we conduct the first empirical study to explore the above research questions. In particular, we analyzed 200 real workarounds that were collected from 81 Apache projects. Our results lead to eight findings and answers to all the above questions. For example, if bug reports are resolved as workarounds, their problems often either arise in external projects (40%) or reside in programming environments (23.5%). Although the problems of some workarounds (38.5%) reside in the project where they are reported, it is difficult to fix them fully and perfectly. Our findings are useful to understand workarounds, and to improve software projects and issue trackers.

Keywords Workaround · Bug fix · Empirical study

1 Introduction

Software maintenance is expensive, and some studies [103, 139] report that around 90% of software life cost is related to software maintenance. In software main-

Aoyang Yan, Hao Zhong, Daohan Song and Li Jia are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.
Hao Zhong is the corresponding author.
E-mail: zhonghao@sjtu.edu.cn

tenance, a major task is to fix bugs [121,106]. To manage the process of fixing bugs, issue tracker systems (*e.g.*, JIRA [1]) are widely used in both open source and commercial projects. In an issue tracker, Zeller [151] and Jeong *et al.* [115] introduce that a bug report can be filed, diagnosed, assigned, and fixed. In the standard process (see Section 2.1 for details), a bug report typically is resolved as *fixed*, *invalid*, *duplicated* or *won't fix*. Although the reporter of a bug report can disagree with its resolution, the definitions of their resolutions are clear, and users can make their choices based on these resolutions. For example, if the problem of a bug report is serious and the bug is *fixed*, a user can update to a newer version to avoid the problem.

Although it is less known, a bug report can be resolved as a *workaround* (see Section 2.2 for such an example). By its literal definition, a workaround is a way to resolve or avoid a problem, when the most obvious solution is not possible. After reading this definition, the resolutions of workarounds still raise questions besides what are already reported in bug reports. For example, are the problems fixed or not? If a problem is literally avoided, is it a technical debt [150]? Why are bug reports resolved as workarounds at the first place? Indeed, even the very definition of workarounds is unclear. When the prior studies (*e.g.*, [115]) introduce the work flow of issue trackers, they do not mention workarounds at all. The questions hinder users from making a good decision, even if the user knows that a bug is fixed as a workaround.

To deepen the knowledge on workarounds, in our ICSE 2022 poster [149], we advocate conducting an empirical study. In this study, we collected 200 real workarounds from 81 Apache projects, and our collected workarounds included both API workarounds and other workarounds. Compared with our two-page poster [149], this extended version has the following additional contributions:

1. Cleansed workarounds. In our two-page poster [149], we report that programmers mark two different types of bug reports as workarounds. As a minority, programmers can mark a bug report as a workaround, if its problem is already fixed (*e.g.*, in a newer version). Some researchers criticize that these workarounds must be removed. In this work, we follow their advice, and present the results after removing these workarounds.
2. More protocol details and examples. For all the research questions, we present their detailed protocols and more illustrative examples.
3. Detailed findings and interpretations. We present our detailed findings and actionable interpretations.

Our study explores the following research questions:

– **RQ1. What are symptoms of bugs, if they are fixed as workarounds?**
Motivation: The answers are useful for programmers to fix bugs. If a bug report describes a similar symptom that has been fixed as a workaround, programmers can consider to fix this bug as a workaround.
Answer: When bugs are fixed as workarounds, their symptoms are mostly crashes (40.5%) and unexpected behaviors (33.5%) (Finding 1). Crashes are resolved as workarounds, often when their problems reside out of the project (Finding 8), and unexpected behaviors are resolved as workarounds, often when they become technical debt (Finding 5).

- **RQ2. Why are workarounds introduced?**
  Motivation: The symptoms alone are insufficient for programmers to determine whether a bug shall be fixed as a workaround. To make correct decisions, programmers need to understand the causes of workarounds. The causes explain why a bug report is not resolved by the standard process.
  Answer: Finding 3 shows that 40% of workarounds are caused by problems in external projects, and 23.5% of workarounds are caused by programming environments. Although 38.5% of workarounds do not involve other projects, they are difficult to be fully fixed (Finding 2).
- **RQ3. How do workarounds repair bugs?**
  Motivation: If programmers cannot fully fix a bug, they may be curious about how other programmers handle similar bugs. To answer this research question, we analyze the repair patterns of workarounds. Our answers can be useful for programmers to learn how to resolve bugs as workarounds.
  Answer: Finding 4 shows that 41% of workarounds modify the interfaces between projects, and their problems are avoided by switching libraries (*e.g.*, a newer version) or switching the way to call APIs (*e.g.*, modifying API calls). Finding 5 shows that 40.5% of workarounds modify the project where bugs are reported. Among them, most workarounds modify non-source files (*e.g.*, settings), and the remaining workarounds are often still technical debt after repairs. When a problem arises in the programming environment, as workarounds, operating systems and underlying techniques can be modified (Finding 6). A few workarounds have no repairs for two different reasons: some (4.5%) are already fixed and some (3.5%) will not be fixed (Finding 7).
- **RQ4. What are the associations among symptoms, causes, and repairs of workarounds?**
  Motivation: The answers are useful for programmers to understand the relations among symptoms, causes, and repairs of workarounds. When they learn a symptom of a workaround, programmers can understand its most likely causes, and choose the most feasible repairs.
  Answer: The associations of symptoms, causes and repairs can be non-intuitive and inconsistent with a prior study [120] (Finding 8). For example, switching versions is a more frequent API workaround than deep copying.

Finding 5 shows that 4% of workarounds are technical debts. Typically, the technical debts of a project refer to the problems of this project, and such problems can be fixed in its future versions. In contrast, workarounds can be cross-projects. For example, when they encounter compiler bugs, programmers have to bypass them with workarounds [155], and a bug report from `NixOS` [94] describes such a workaround. This workaround is not a technical debt, and it can be removed, only after the corresponding compiler bug is fixed.

2 Preliminary

This section introduces the standard process of handling a bug (Section 2.1), and that of a workaround (Section 2.2).

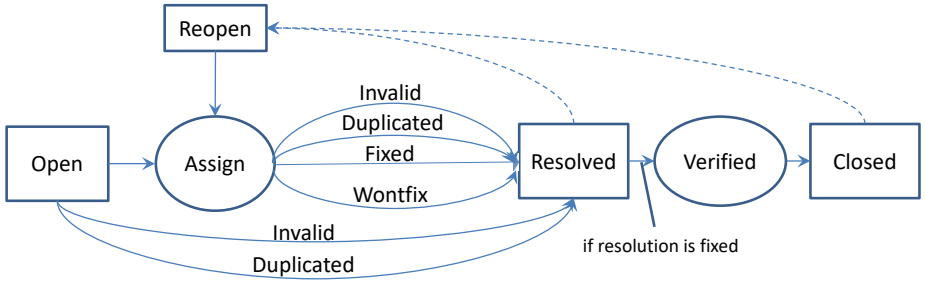

Fig. 1: The life cycle of a bug report

2.1 Standard Process

The standard process of handling a bug report is intensively studied. For example, Jeong *et al.* [115] report the process of Bugzilla [2]. As our analyzed workarounds come from another bug management system called JIRA, we revised the process according to JIRA as shown in Figure 1. In this figure, a rectangle indicates a status of a bug report; an oval indicates an action taken by a programmer; and an arrow denotes a transition from a status to another. A word above an arrow denotes a resolution. A dotted line denotes that a transition may not happen. For example, only a few bug reports are reopened, and this type of transitions is uncommon.

As shown in Figure 1, initially, when a programmer or user reports a bug to JIRA, a corresponding bug report is created and its status is set to *open*. Next, after some inspections, if the bug report is considered as *invalid* or *duplicate*, it will be marked as *resolved*, otherwise it will be assigned to an assignee. The assignee will resolve the bug, and based on how the bug is handled, the *resolution* of the bug can be marked as *invalid*, *duplicate*, *fixed*, or *won't fix*. A *fixed* bug report can be marked as *closed*, after other programmers inspect it. If a bug report is not fully fixed, it can be reopened. At any step of handling a bug report, programmers could participate in the discussion and make their contributions such as patches, links to other related issues, and pull requests.

2.2 Workaround Process

In this study, we define workarounds as follows:

Definition 1 A workaround is a bug report whose problems are bypassed.

As an example of workarounds, we next introduce `BEAM-6460` [3]. `Beam` [4] is a framework that handles batch and streaming data. It is built on `Flink` [5], a framework that handles data streams. On 17th Jan. 2019, Max* (the full name is hidden for privacy) filed a bug report [3] of `Beam` as shown in Figure 2. The bug report is slightly modified to fit space limit. In this figure, we mark the sentences that describe its symptom: when the application restarts due to a failure, old objects may not be garbage collected. Max* determined that this is not a `Beam` bug, but `Flink` leaks when the application restarts. Although the bug cannot be fully

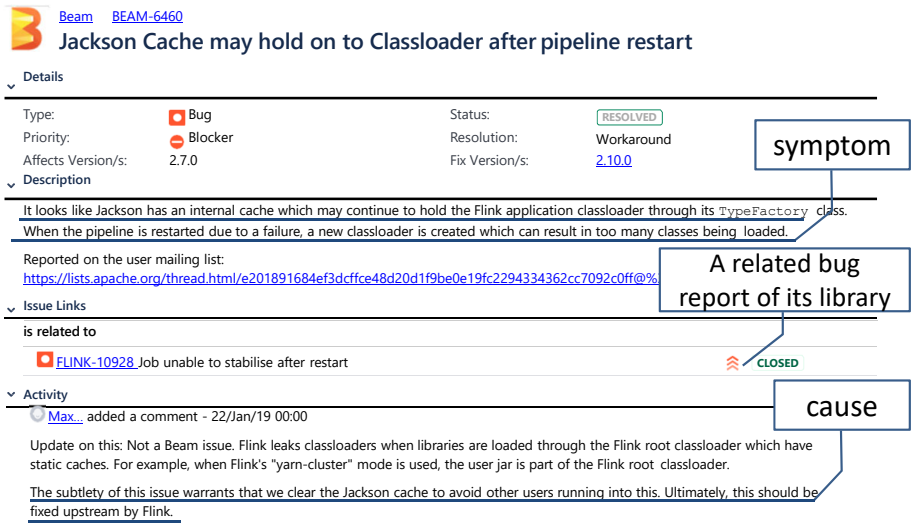

Fig. 2: A workaround

fixed, as a workaround, Max* submitted a pull request [6] as shown in Figure 3. Its message in Figure 3a says that the `Flink` classloader is not garbage collected, if an object of the classloader is still referenced. As `Beam` programmers cannot directly modify `Flink`, Max* determined to delete the cache of `Beam`, when the application restarts. For example, Figure 3b shows one of the buggy locations. The `invokeTeardown` method fails to clean all the objects, and can throw exceptions due to the mentioned problem of `Flink`. As the problem cannot be perfectly fixed, Max* added the `deleteStaticCache` method after the `invokeTeardown` method to delete the caches. To fully fix the problem, as shown in Figure 2, another bug report has been filed to `Flink` [7]. The programmers of `Flink` determined that this problem is caused by the garbage collection of JVM. As it is difficult to fix the problem, the bug has not been fixed since it was reported. Although it was finally marked as resolved, from its code repository [8], we do not find code changes that are related to the `Flink` bug report. We inspected the latest version of the `close` method. Compared Figure 3d with Figure 3c, although it calls a different method, it still relies on some workarounds to handle this problem. In this example, the `Beam` bug report is resolved as a workaround, because its problem resides in `Flink`. Although its repair is imperfect, programmers have to live with similar repairs because the problem of `Flink` resides in JVM and is difficult to fix.

Many bug reports are resolved as workarounds, because their problems are out of scope. To understand the scope of a bug report, we introduce the following terms. If a bug report is filed to a project, we call this project as *my project*. In the above example, `Beam` is my project, since the bug report is filed to `Beam`. *My library* is a project that is called by my project. In the above example, `Flink` is my library, *i.e.*, a library of `Beam`. *My client* is a project that calls my project. Based on the scope, we classify the causes of workarounds (see Section 5.2 for details). Please note that the role of a project can change according to call relations. For

example, as a Java project, `Flink` calls the APIs of `J2SE`. In this call relation, `Flink` is a client of `J2SE`.

2.3 API Workarounds and Workarounds in Issue Trackers

If a workaround occurs in a library, it can lead to far-reaching impacts on its downstream projects, and a workaround on an API can reveal problems of its upstream projects. As API workarounds have notable impacts on programmers, researchers have conducted some early studies on API workarounds. Bogart *et al.* [101] report that users can intentionally modify or bypass a problematic API as a workaround. Lamothe and Shang [120] summarize four patterns on API workarounds. Workarounds in issue trackers and API workarounds are different in their scopes. For example, in the study of Lamothe and Shang [120], they analyzed Stack Overflow posts, but our study analyzes workarounds in issue trackers. As a result, our results are more useful for programmers to handle issue reports. As issue reports present more types of workarounds, the findings of our study enrich the knowledge of the prior studies. For example, Lamothe and Shang [120] summarize four patterns on API workarounds. As introduced in Section 5.4, from issue reports, we have found more API workarounds that were not reported by Lamothe and Shang [120]. In practice, Herzig *et al.* [113] show that programmers can wrongly classify issue reports. As workarounds are less known, many workarounds can be resolved as other resolutions. Our study improves the awareness of workarounds, and is useful for programmers to correctly handle their issue reports.

3 Related Work

Our work is related to the following research themes.

3.1 Empirical Study on Bug Report and API Library

Various empirical studies are conducted to understand bug reports and their handling process. Anvik *et al.* [97] introduce the standard life cycle of bug reports. Bettenburg *et al.* [99] show the values of duplicate reports. Guo *et al.* [110] summarized the factors (*e.g.*, reputations of programmers) to determine which bug shall be fixed. Some studies analyze how to improve the quality of bug reports [98, 157], because poor bug reports can result in too many reassignments [111]. Lin *et al* [125] compare two bug triage approaches. Xia *et al.* [148] made an empirical study on bug report field reassignment. For bug fix, there have been some empirical studies on bug fixing time [152,153], bug fixing in open source projects [108,100]. Li and Zhong [123] analyzed the impacts of obsolete bug fixes. Our study focuses on the workarounds of bug reports, which are ignored by the above studies.

There are numerous empirical studies on API libraries. These studies explore the knowledge on API documents [127,104,140], API deprecation [136], API evolution [114,140], the impact of API changes on software quality [145], the impact of API changes on Stack Overflow discussions [146], parallel API libraries [131],

specific API libraries [128, 116], API libraries in specific languages [134], mining API properties [156], and API learning obstacles [138]. Lamothe *et al.* [119] conduct a systematic survey on the evolution of APIs. Our study reveals that many workarounds are related to API calls, and it can be feasible to reduce workarounds by improving the designs of APIs.

3.2 Technical Debt

Technical debts have been intensively studied. Potdar and Shihab *et al* [133] analyze the technical debts from four large open source projects. Kazman *et al* [117] analyze the architectural roots of technical debts. Tang *et al* [142] analyze refactorings and technical debts in machine learning systems. Ramasubbu and Kemerer [135] propose an approach to manage technical debts. Vetrò and Antonio [147] propose a static-analysis approach to identify technical debts. Tom *et al* [144] conduct a systematic review on technical debts. Our study reveals that some workarounds are caused by technical debts.

3.3 Mining API Rule

Nguyen *et al.* [130] mine API rules based on graphs, and some researchers [96, 109, 132] mine automata for API rules. Robillard *et al.* [137] explain the above two approaches are equivalent. Li *et al.* [124] extract function call pairs, and Engler *et al.* [105] consider bugs as deviant behavior from frequent call sequences. Related research can be reduced to mining sequential patterns [95]. Ernst *et al.* [107] detect invariants for API rules, and Lorenzoli *et al.* [126] use frequent call sequences and invariants, to extract models of functional behavior. Dallmeier *et al.* [102] leverage test cases for API rules mining. Compared with Lamothe and Shang [120], our study reveals more API workarounds, and analyzes software workarounds as a general problem.

4 Methodology

This section introduces our dataset (Section 4.1) and protocols (Section 4.2).

4.1 Dataset and Preprocessing

To collect the dataset, we search Apache JIRA [1] for bug reports whose resolutions are workarounds and statuses are *resolved* or *closed*. In other words, we select closed bug reports that are resolved as workarounds by their developers. We select Apache projects, because they have an easy-to-use search interface. In particular, we collect our workarounds through a query:

`https://tinyurl.com/yxvxjd2f`.

In total, we collected 221 workarounds, and to ensure the diversity of the dataset, these workarounds were collected from 88 Apache projects. The size of

[BEAM-6460] Remove cached class references upon start/shutdown The Flink Classloader can only be garbage collected if the classes it loaded are not referenced anymore. Users have reported that old classes leaked through Jackson's TypeFactory ...

(a) message

```
public void close() throws Exception {
    doFnInvoker.invokeTeardown();
}
```

(b) One buggy location

```
public void close() throws Exception {
    try {
        doFnInvoker.invokeTeardown();
    } finally {
        FlinkClassloading.deleteStaticCaches();
    }
}
public class FlinkClassloading {
    public static void deleteStaticCaches() {// Clear
        cache to get rid of any references to the Flink
        Classloader
        // See https://jira.apache.org/jira/browse/BEAM
            -6460
        TypeFactory.defaultInstance().clearCache();
    }
}
```

(c) Fixed code

```
public void close() throws Exception {
    try {
        metricContainer.registerMetricsForPipelineResult
            ();
        Optional.ofNullable(doFnInvoker).ifPresent(
            DoFnInvoker::invokeTeardown);
    } finally {
        Workarounds.deleteStaticCaches();
    }
}
```

(d) The latest code

Fig. 3: The pull request of Max*

our dataset is comparable to those of other related empirical studies. For example, Zhang *et al.* [154] analyzed 175 tensorflow [9] bugs.

Figure 4 shows our dataset. The Apache Foundation classifies its projects into several categories [10]. An Apache project can be classified to multiple categories. For example, `JCLOUDS` [11] is classified to `cloud` and `library` because it provides interfaces to call various cloud services (*e.g.*, Amazon and Azure). The result shows that our workarounds come from all categories of Apache projects, which highlights the diversity of our dataset.

Herzig *et al.* [113] find that researchers and programmers have different definitions on the types of issue reports. After our inspection, we find that even programmers themselves have different definitions on workarounds. In particular, we find that in total, the problems of 21 workarounds are already fixed when they are reported. For example, the problem of a bug report [12] is fixed in newer versions, and the problem of another bug report [13] is fixed when repairing other

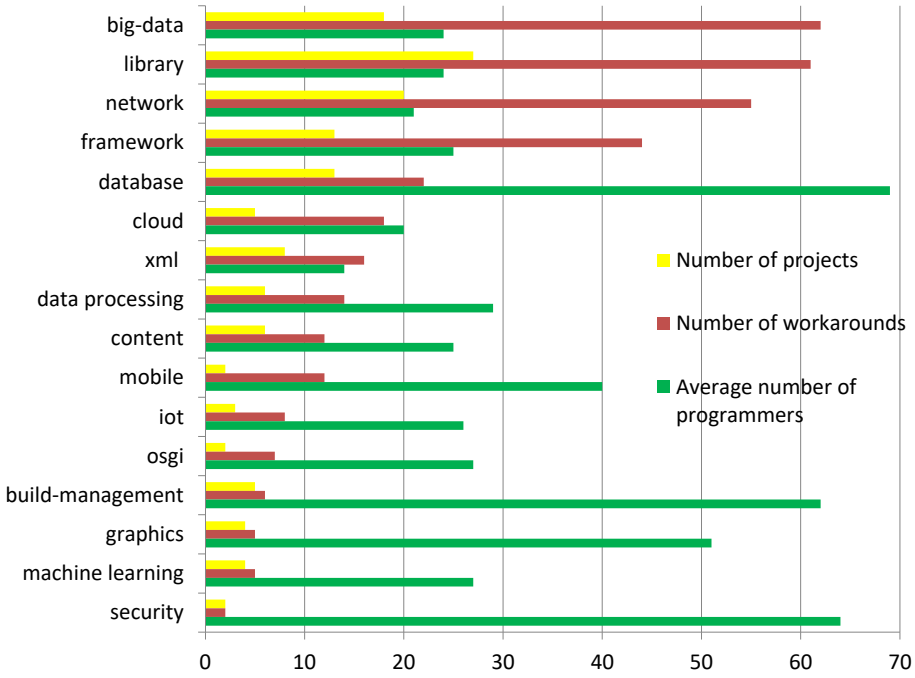

Fig. 4: The distribution of our selected workarounds.

related issues. Although they are marked as workarounds by programmers, in our previous submissions, multiple reviewers insist that these workarounds shall be removed from our dataset, since the resolutions of these bugs are somewhat inconsistent with the literal definition of workarounds. Following their suggestions, in this study, we remove the 21 workarounds, and conduct our study on the remaining 200 workarounds.

4.2 Overall Protocol

In our study, we manually inspected all the 200 workarounds with the following steps: First, for each bug report, we read the website of the project to understand its functionality and users (*e.g.*, programmers or end users). This analysis is useful to understand the scope of bug reports. Second, we read the title, description and discussions of a bug report to understand its symptoms. Third, we read the discussions of a bug report to understand its causes and repairs. A programmer marks a bug report as a workaround, and the comments from this program are typically useful to understand why a bug is resolved as a workaround. If bug reports have related issue reports and pull requests, we further read their related issue reports and pull requests. For each bug report, we checked its code from its github repository and searched for commits that resolve a bug with its issue number. If such commits are found, we read their code changes to determine their types of causes and repairs. The categories of symptoms are predefined by prior

studies [154,141,143,116], but the categories of causes and repairs are defined by ourselves, since no prior study has built such taxonomy for workarounds. Following this protocol, we inspected the bugs independently, and compared the results for differences. If any result was inconsistent, we discussed it on our group meetings and through emails.

Researchers typically apply two types of sorting studies [129]. In an open card sorting study, there is no predefined categories, researchers classify cards according to their understanding, and in a closed card sorting study, researchers classify cards according to predefined categories. As the symptoms of bugs are already intensively explored, in the first research question, we apply the closed card sorting study, and our symptoms are predefined in the prior studies [154,141,143, 116]. As the cases and repairs of workarounds are rarely explored, we apply the open sorting study in the other research questions. Our study is conducted in two rounds. In the first round, the third and the fourth authors build the taxonomy, and in the second round, the first author rebuild the taxonomy. The second author plays as the reviewer for both rounds. Krippendorff's α testing is widely used to measure the inconsistencies among items [118,120]. This value is between zero and one, where zero denotes a random chance and one denotes a perfect agreement. Initially, between the two rounds, the α values of RQ1, RQ2 and RQ3 are 0.874, 0.929, and 0.935, respectively. Although Krippendorff's α testing presents a statistic confidence, it still leaves such inconsistencies to readers, and it is ambiguous to calculate the percentages in the presence of inconsistent cases. In this study, we tried to resolve all inconsistencies. If we could not come to an agreement, we contacted their programmers by sending emails or directly discussing on its bug report. For example, a bug report [14] is difficult to understand. It does not have an informative title; its description contains only a stack trace; and its discussions are brief. Although the bug report presents two related pull requests, the relations between the bug report and the two pull requests are unclear to us. As a result, it was difficult for us to fully determine the causes of the workaround. We asked programmers who handle the bug report, and a programmer named Makoto Yui confirmed that this bug is caused by the memory requirements of `Spark` [15] and they have fixed this problem.

In each RQ, we analyzed the 200 workarounds according to its corresponding protocol. In RQ1, RQ2, and RQ3, after we determine the categories of all workarounds, the percent of a category A is calculated as follows:

$$ratio_A = \frac{N_A}{N_{All}} \tag{1}$$

where N_A is the number of workarounds in category A, and N_{All} is the number of all workarounds, *i.e.*, 200. In RQ1, a workaround is put into exactly one category, so the sum of the percentages is one. In RQ2 and RQ3, several workarounds are classified into multiple categories, so the sums of the percentages can be more than one.

5 Empirical Result

Figure 5 shows the overview of our identified categories. The grey bar behind each category denotes its percentage, and the links denote their associations. We next

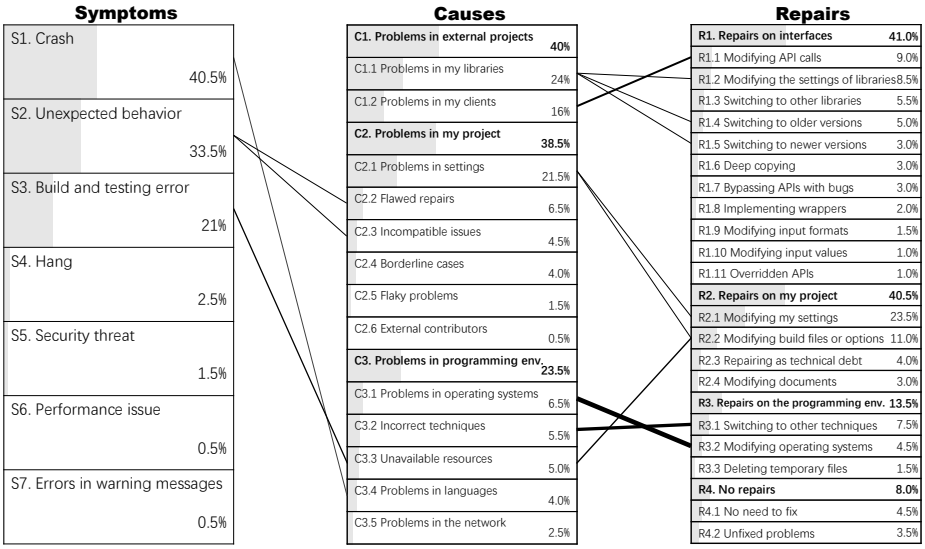

Fig. 5: The distributions and associations of symptoms, causes and repairs.

introduce our classified symptoms, causes, and repairs. More details are presented on our website:
`https://github.com/tetradecane/Workaround_journal_website`

5.1 RQ1. Symptoms and Standard Process

5.1.1 Protocol

In this research question, we classify bug reports by their symptoms. A symptom of a bug report is the observable buggy behavior of its described bug. As this research question explores which symptoms are likely to be resolved as workarounds, our symptom taxonomy has to be aligned with those of the prior studies [154, 141, 143, 116]. As they did, we read a bug report to identify its symptom. In particular, from a bug report, we inspected its title, description, and comments. Based on their taxonomies [154, 141, 143, 116], when we inspected bug reports, we located sentences that are useful to identify the category of a bug (*e.g.*, crashes, hangs, and unexpected behaviors). For example, in Section 2.2, we highlight the symptom sentence of the sample bug report, because this sentence describes an unexpected behavior of garbage collection.

5.1.2 Result

In the standard process, if a bug has a symptom, programmers identify the root cause of the symptom, and typically modify code to fix the bug. As a comparison, for each symptom, we introduce examples in this process.

S1. Crash (81/200, 40.5%). A crash occurs, when a program fails to terminate normally. For example, a workaround [16] describes a crash: “DNS Packets

with dns.flags.rcode=1 cause ml_ops.sh to crash". In the standard process, programmers determine that a crash [17] is caused by a wrong way to retrieve values from a table, and fix the code to retrieve the correct values.

S2. Unexpected behavior (67/200, 33.5%). An unexpected behavior occurs, when a user or programmer experiences deviation from expected behaviors. The symptom of a workaround [18] is an unexpected behavior, because the report has a sentence: "I'm getting some extremely strange behavior when trying to extract features for a learning to rank model". In the standard process, programmers determine that an unexpected behavior [19] is caused by the wrong calculation of two values. To fix the problem, the correct value is calculated [20].

S3. Build and testing error (42/200, 21%). Build and testing errors occur in the building and testing process. Apache provides some Continuous Integration (CI) tools to support the automation of building and testing. If these tools report errors, we also put these errors into this category. For example, the symptom of a workaround [21] is a build error: "Builds are failing in pipeline due to SSL locator tests failing". In the standard process, a bug report [22] complains that a test case wrongly failed on Java 7, and programmers determine it is caused by missing vendor names. The bug is fixed by adding those missing names [23].

S4. Hang (5/200, 2.5%). A hang is a situation when a program does not stop nor respond. For example, the symptom of a workaround [24] is a hang: "... noticed that a job reading avro files would have some tasks that never finish. Looking at the threads they got stuck in". In the standard process, programmers determine that a hang [25] is caused by a logic error in the `URI` class, and the error is fixed to resolve the hang.

S5. Security threat (3/200, 1.5%). A security threat presents the vulnerability on software security issues. For example, the symptom of a workaround [13] is a security threat, because a user can improperly change the permission of other users:" Lets suppose user1 has created Notebook 9 and he has not changed the note permissions. When user2 logins, as the Notebook 9 is visible to user2, user2 can change the permissions of the user1 note (Notebook 9)". In the standard process, a bug report [26] complains that a user can see the personal data of other users. This security threat is caused by too lax default permissions, and it is fixed.

S6. Performance issue (1/200, 0.5%). If a program does not respond in expected time or consumes too many computational resources, we determine it as a performance issue. For example, the symptom of a workaround [27] is a performance issue: "REST search queries make Ranger incredibly slow". In the standard process, programmers determine that a performance issue [28] is caused by setting the size of a table as a constant. To fix the bug, programmers reimplement the table to allow it to increase its size.

S7. Errors in warning messages (1/200, 0.5%). This type of errors occurs, when warning messages are wrong or producing warning messages leads to errors. For example, a workaround [29] includes a wrong warning messages: "It seems like for each integration test run we produce about 400MB of logs. This is way too much for one run". In the standard process, programmers determine that an error in warning messages [30] is caused by a missing forward slash ("/") in `web.xml` file. Adding missing required slashes fixes this bug.

Li *et al.* [122] report that the symptoms of most bugs are incorrect functionalities (64.3% to 69.4%) and crashes (14.3% to 19.1%). Our unexpected behaviors

correspond to their incorrect functionalities. We find that incorrect functionalities and more crashes are likely to be resolved as workarounds:

Finding 1 *When bugs are resolved as workarounds, their symptoms are often crashes (40.5%) and unexpected behaviors (33.5%).*

5.2 RQ2. Causes of Workarounds

5.2.1 Protocol

In this research question, we determine the causes of a workaround. As they have never been explored, in this and the following research questions, we follow different analysis methodologies from the prior ones [154,141,143,116]. The prior ones analyze the causes of bugs. The cause of a bug explains why this bug occurs. In our study, we do not analyze the causes of bugs, but the causes of workarounds. In particular, given a bug report, we analyze why programmers do not choose the standard process as described in Section 2.1, but would rather repair the bug as a workaround (*e.g.*, the one as described in Section 2.2). In particular, from a bug report, we searched for the sentences that explain why a bug has to be fixed as a workaround, and the sentences that explicitly mention workarounds. For example, as shown in Figure 2, the bug report is fixed as a workaround, because programmers believe that this problem shall be fixed by its library. Although the bug is reported to `Beam`, the problem resides in its library, `Flink`. Although `Beam` programmers have bypassed the problem, the problem itself is resolved imperfectly. As the programmer said, the problem shall be fixed by the upstream, `Flink`. In this example, the bug is fixed as a workaround, because its problem resides in the library, and we put it to *C1.1 Problems in my libraries*.

5.2.2 Result

The causes of workarounds are as below:

C1. Problems in external projects (80/200, 40%). These bug reports are resolved as workarounds, because the programmers of a project typically cannot fix the problems external of the project.

C1.1 Problems in my libraries (48/200, 24%). Although a bug report is submitted in a project, it is caused by the libraries of this project. As it is infeasible to repair a problem in libraries, the bug report is often marked as a workaround. For example, `Thrift` [31] is a framework for developing scalable cross-language services, and it calls `hspec-core` [32] as a library. A bug report [33] complains about a build failure when compiling with `hspec-core 2.4.0`. Programmers determine that `hspec-core 2.4.0` has a bug, and limit `hspec-core` to a version below `2.4.0` to avoid the build failure.

C1.2 Problems in my clients (32/200, 16%). Although a bug is reported to a project, its problem does not reside in the project, but in the clients of the project. For example, `Log4j` [34] is a logging service, and is called by other projects like `IBM WebSphere`. A bug report [35] of `Log4j` describes a crash when a `WebSphere` server calls `Log4j`. Although the problem is reported to `Log4j`, it turns out to be a

bug in `WebSphere`, and it is fixed as a `WebSphere` bug [36]. In some cases, programmers of clients do not fully understand the APIs of a project, and report their problems to the project. For example, `Spark` [15] presents a `BigDecimal` type to define decimal values. A bug report [37] complains that the multiplication of two `BigDecimal` values returns `null`. A programmer of `Spark` explains that `Spark` implements a technique to detect overflow and the `null` value indicates a detected overflow.

C2. Problems in my project (77/200, 38.5%). Programmers cannot fully or perfectly fix some problems of their projects, and mark their bug reports as workarounds. Some problems are related to technical debt [150].

C2.1 Problems in settings (43/200, 21.5%). If the problem of a bug report is caused by wrong settings, programmers can recommend correct settings, and mark the bug report as a workaround. For example, a bug report [38] of `Spark` [15] complains that `Spark` applications are invisible from `JSON` APIs. According to a document [39] in the `Spark` website, `spark.eventLog.enabled` must be set as `true` to enable `JSON` APIs, but the reporter did not set it as true. In a comment of this bug report, a program explains that it takes significant programming effort to fully fix the bug, but it is simple to fix the bug as a workaround by turning on the `eventLogging` setting.

C2.2 Flawed repairs (13/200, 6.5%). Although the problem of a bug report seems to be fixed, the fixed code can be flawed, and can produce wrong results in some cases. For example, `Mesos` [40] is a distributed cluster manager. A bug report [41] of `Mesos` complains that a test case wrongly fails. A programmer determines that a thread does not terminate in the expected time. As a workaround, the programmer changes the expected time from 5 seconds to 10 seconds. He believes that the repair is flawed and recommends a better repair for this problem in his comments.

C2.3 Incompatible issues (9/200, 4.5%). Although the problem of a bug report is fixed, the fixed code can introduce incompatible issues. For example, `OpenJPA` [42] is a persistence framework. A bug report [43] of `OpenJPA` complains that it maps the `double` type in Java to the `NUMERIC` type in `HSQLDB`. A straightforward repair is to change the mapping from the `NUMERIC` type to `DOUBLE`, but the modification can introduce a backward incompatible issue. Instead, programmers reject this modification, and live with the problem.

C2.4 Borderline cases (8/200, 4%). A bug report can be triggered by borderline cases (*e.g.*, special inputs). Instead of fixing buggy code, programmers can avoid such borderline cases and mark the corresponding bug report as a workaround. For example, `Spot` [44] can detect security threats in network flows and packets. A bug report [16] of `Spot` describes a crash when it scans a specific type of packets. Instead of fixing the buggy code, programmers modify its setting file and ignore such packages.

C2.5 Flaky problems (3/200, 1.5%). The problems of some bug reports are flaky and difficult to reproduce. For example, `Impala` [45] is an SQL query engine. A bug report [46] of `Impala` complains about a hang in its build process. However, programmers cannot reproduce it, and the problem disappears after restarting the build task.

C2.6 External contributors (1/200, 0.5%). Before a bug report is fixed, users can implement their own tools to fix the problem of the bug report. As these tools are not included in the reported project, the bug report is marked

as a workaround. For example, a bug report [47] requests a migration tool, but programmers ignore this problem after it is reported. To handle the problem, the reporter implements and releases his own tool on Github [48], and the bug report is marked as a workaround.

The above causes lead to the following finding:

Finding 2 *In 38.5% of workarounds, programmers modify their own projects, but their repairs are imperfect (*e.g.*, flawed repairs 6.5%), difficult to reproduce (*e.g.*, flaky problems 1.5%), modifications on non-source code (e.g., settings 21.5%), or solved by external contributors.*

C3. Problems in programming environments (47/200, 23.5%). The problem of a bug report occurs in the programming environments of a project. Although modifying the programming environments of a project can avoid the problems, the modifications must be applied wherever the project is deployed. As their repairs are imperfect, their bug reports are marked as workarounds.

C3.1 Problems in operating systems (13/200, 6.5%). The problem of a bug report resides in operating systems. As it is difficult to fully fix such a bug, its report has to be fixed as a workaround. For example, a bug report [49] of `Mesos` complains about a test case failure. Programmers determine that the failure is caused by a Linux file service called `LXCFS`. As a workaround, they disable the service.

C3.2 Incorrect techniques (11/200, 5.5%). This type of problems occurs, when programmers choose incorrect techniques. For example, a bug report [50] of `Cordova` [51] complains about a performance degradation, when the reporter uses a `Javascript` framework, `jQuery`, to set the layout of a mobile application. A programmer of `Cordova` explains that `Javascript` is not a good choice, and recommends the reporter to use `media queries` [52].

C3.3 Unavailable resources (10/200, 5%). The problem of a bug report can be caused by unavailable or missing resources (*e.g.*, broken URLs). For example, `NetBeans` [53] is an integrated development environment. A bug report [54] describes a build failure, due to a broken `Maven` dependency.

C3.4 Problems in languages (8/200, 4%). The problem of a bug report can be caused by the programming languages of its project, so it is fixed as a workaround. For example, `Solr` [55] is an open source enterprise search platform. A bug report [56] of `Solr` complains about a build failure, when the wildcards such as "*" and "**" are used to load classes. Programmers determine that Java 9 changes its way to scan class paths, and the modification causes the failure.

C3.5 Problems in the network (5/200, 2.5%). The problem of a bug report resides in the underlying network. For example, `JCLOUDS` [11] is a cloud toolkit. A bug report [57] complains that an unexpected error message is received, when it works with `Amazon Web Service`. Programmers determine that it is caused by an unidentified problem in the underlying network. As they cannot solve the problem, they have to recover their service when it happens.

The above causes lead to the following finding:

Finding 3 *The problems of over half of workarounds are in external projects (40%) or reside in programming environments (23.5%).*

5.3 RQ3. Repairs in Workarounds

5.3.1 Protocol

In this research question, we analyze how the problem of a workaround is repaired. For a given bug report, we analyze its pull requests and commit histories, if they are available. For the example in Section 2.2, the problem resides in `Flink`, a library of `Beam`. As shown in Figure 3, to fix the problem, programmers bypass the API method with the problem. When it throws exceptions, they implement code to handle the problem. As a result, we classify its repair to *R1.8 Bypassing APIs with bugs*. If we cannot find corresponding pull requests or commit histories, we read the comments to understand how the programmers repair the problems. Indeed, different from other bug reports, the problem of a workaround is often in external projects. To fully understand the repair of a workaround, we even read the bug reports and modifications of other projects. In this example, after we read the bug report of `Flink` [7] and its modifications, we confirm that this problem resides in `Flink`.

5.3.2 Result

The repairs of workarounds are as follows:

R1. Repairs on interfaces (82/200, 41%). The problem of a bug report lies in a library. Before the problem is fixed in that library, as workarounds, programmers change their ways to call APIs. The category has two scenarios. First, a bug is reported by the client programmers of a library, but the programmers of the library do not fix the bug. In this scenario, the client programmers have to repair their client code. For example, `ActiveMQ Artemis` [58] is a messaging framework. A bug report [59] complains about a connection failure. A programmer identifies that the problem is recurring [60]. To resolve this problem, he recommends downgrading the library of clients from `2.6.2` to `2.6.1`. Second, a bug is report to a project, but the problem lies in a library of the project. If the library does not fix the problem, the programmers of this project have to repair their own code. We next present such an example.

R1.1 Modifying API calls (18/200, 9%). Client code can produce unexpected results because it wrongly calls APIs. After programmers identify the problem, they will update their client code to call correct APIs, and these modifications are considered as workarounds. For example, `Synapse` [61] is a framework for service management and integration. A programmer named amit complains that the HTTP header is not as expected [62]. Later, amit leaves the correct API calls in a comment of the bug report.

R1.2 Modifying the settings of libraries (17/200, 8.5%). The problem of a bug report can be relieved by modifying settings. For example, in a bug report [63], `Solr` hangs when `JDBC` is stuck in the middle of a read, and increasing `oracle.jdbc.ReadTimeout` can relieve the problem.

R1.3 Switching to other libraries (11/200, 5.5%). A library has problems, but there are other libraries as replacements. If a report describes such problems, programmers can recommend the alternative libraries, and mark the bug report as a workaround. For example, `NiFi Registry` [64] is a project to store

and manage shared resources. A bug report [65] of `NiFi Registry` complains about a crash with `JRE`. A programmer of `NiFi Registry` explains that `NiFi Registry` must call `JDK` rather than `JRE`, and resolves this problem.

R1.4 Switching to older versions (10/200, 5%). The problem of a bug report is recurring. When a problem occurs in a library of the project, before the problem is fixed, programmers can recommend switching to older versions of the library. For example, as we introduced the cause of *problems in my libraries*, the problem of a bug report [33] is caused by a buggy version `2.4.0` of library `hspec-core` [32]. Programmers avoid the problem by limiting library `hspec-core` to a version below `2.4.0`.

R1.5 Switching to newer versions (6/200, 3%). The problem of a bug report is already fixed in a newer version. When such a problem is reported to a project, programmers can recommend switching to newer versions of the project. For example, `Openmeetings` [66] is a video conference application, and `Moodle` [67] is a learning management system. `OpenMeetings` has a module named `openmeetings-moodleplugin` [68] to connect to `Moodle`. Here, the module is developed in a separated project, and can be considered as a library of `Openmeetings`. A bug report [69] of `OpenMeetings` complains about a connection failure. A programmer of `OpenMeetings` recommends using a newer version of `openmeetings-moodle-plugin`, and thus resolves this problem.

R1.6 Deep copying (6/200, 3%). If library developers do not fix problems in a library, programme modify library code directly. For example, `Maven` [70] calls `jansi` [71] as a library to print colorful texts. A bug report [72] complains that `Maven` does not print colorful texts when it works with NetBeans. To fix the problem of `maven`, a programmer copied the source files of `jansi` and modified these source files directly [73].

R1.7 Bypassing APIs with bugs (6/200, 3%). If the problem of a library is too difficult to solve, programmers can choose to bypass the related APIs of the library. For example, `Nemo` [74] is a data processing project, and it calls `Guava` as a library. A bug report [75] of `Nemo` complains about a serialization exception that is related to `Guava`. The programmers of `Guava` are working on this problem, but the problem is difficult to solve, because it is caused by an unresolved Java problem [76]. As a workaround, when the exception is caught, programmers print corresponding class names to warn users, but leave the problem unresolved.

R1.8 Implementing wrappers (4/200, 2%). When a library does not work as expected, programmers can implement a wrapper of problematic APIs to obtain their desirable results. For example, `zookeeper` [77] is a library of `Solr` [55]. A bug report [78] complains that `Solr` fails to store large files. While `Solr` calls the APIs of `zookeeper` to store files, programmers determine that `zookeeper` is not suitable to store large data. To fix the problem, programmers implement a wrapper for `zookeeper` [79]. The metadata of the file are still stored in `zookeeper`, but its contents are stored externally.

R1.9 Modifying input formats (3/200, 1.5%). Some problems are caused by wrong input formats. As workarounds, such inputs are modified. For example, `WEEX` [80] is a framework for building mobile applications. A bug report [81] of `WEEX` complains that it does not compile an XML file. A programmer of `WEEX` explains that the XML file shall be rewritten in a specific way to avoid the problem.

R1.10 Modifying input values (2/200, 1%). A bug can be caused by incorrect values. As workarounds, such values are modified. For example, the problem

of a bug report [82] is caused by an illegal name, and its repair is to modify this illegal name.

R1.11 Overridden APIs (2/200, 1%). If a library does not work as expected, programmers can override the problematic APIs of the library. For example, `TinkerPop` [83] is a graph computing framework. In a bug report [84] of `TinkerPop`, a client programmer suggests that the `willAllowId` method shall throw exception when it supports string ids but an id is not a string. However, a programmer of `TinkerPop` disagrees with the suggestion, and asks the client programmer to override the method in his own implementation.

The above observations lead to a finding:

Finding 4 *When libraries cannot be modified to repair problems, as work-arounds, programmers can switch whole libraries (*e.g.*, with an older version 5%), switch APIs with problems (*e.g.*, overridden 1%), or switch the way to call APIs (9%).*

R2. Repairs on my project (81/200, 40.5%). After a bug is reported to a project, workarounds modify the files of the project.

R2.1 Modifying my settings (47/200, 23.5%). The problem of a bug report is caused by wrong settings. As a workaround, the settings are corrected to resolve the problem. For example, a bug report [38] of `Spark` [15] complains that `Spark` applications are invisible from `JSON` APIs. A programmer of `Spark` reminds the reporter of a related document [39]. As the document says, `spark.eventLog.enabled` must be set as `true` to enable `JSON` APIs. The reporter changes the setting of the forked `Spark` project to resolve this problem.

R2.2 Modifying build files or options (22/200, 11%). A project may not compile with its default build files and options. As a workaround, the build files or options are modified according to the new environments. For example, a bug report [85] of `Atlas` [86] complains about a build failure in a forked version. Finally, the reporter changes the build option to solve it.

R2.3 Repairing as technical debt (8/200, 4%). The problem of a bug report still exists, in that its repairs have flaws. These repairs are considered as workarounds because they are technical debt. For example, `SSHD` [87] is a `SSH` library. A bug report [88] of `SSHD` complains about out-of-memory crashes. The reporter proposes to limit the queue of messages. Although the modification solves his problem for now, he is not sure whether it is correct. The programmer of `SSHD` encourages him to submit the pull request. However, after some discussions, they agree that the solution is technical debt, and mark the bug report as a workaround. The discussions show that instead of limiting messages, they are working on a new mechanism to handle the problem.

R2.4 Modifying documents (6/200, 3%). Instead of modifying code, as workarounds, programmers can repair documents. For example, a bug report [89] complains that the default value of a parameter is too small. After some discussions, instead of changing the default value, programmers update the document to explain what the value is and how to change it.

The above observations lead to a finding:

Finding 5 *When repairs happen in a project where a bug is reported, they often modify settings (23.5%), build files (11%), and documents (3%). Meanwhile, the repairs on source files are technical debt (4%).*

R3. Repairs on the programming environments (27/200, 13.5%). Programmers can recommend changing the programming environments to repair bugs.

R3.1 Switching to other techniques (15/200, 7.5%). A framework or library may be not designed to work with specific techniques. When this happens, programmers can recommend the correct techniques as a workaround. For example, a bug report [50] of `Cordova` [51] complains about a performance degradation, when clients use `jQuery` to set the layout of a mobile application. A programmer of `Cordova` explains that `Cordova` supports `media queries` [52], instead of `jQuery`.

R3.2 Modifying operating systems (9/200, 4.5%). Operating systems can be modified to fix the problem of a bug report as workarounds. For example, as we introduced the cause of problems in operating systems, the problem of a bug report [49] is caused by a Linux file service called `LXCFS`, and the problem is fixed by disabling it.

R3.3 Deleting temporary files (3/200, 1.5%). A bug can be caused by cache and temporary files. As workarounds, deleting such files can resolve the problem. For example, a bug report [90] of `NetBeans` [53] complains that its code templates do not work. Later, the reporter realizes that deleting cache files resolves the problem.

Finding 6 *As workarounds on programming environments, programmers can recommend switching to other techniques (7.5%), modifying their own operation systems (4.5%), or deleting temporary files (1.5%).*

Comparing with the results in Section 5.2, we find more problems on programming environments than repairs on programming environments. It can be repaired at other locations (*e.g.*, source files), and thus a repair is more general than modifying programming environments.

R4. No repairs (16/200, 8%). Programmers do not repair source files to resolve these workarounds.

R4.1 No need to fix (9/200, 4.5%). The problems of these workarounds are unnecessary to fix, and no further modifications are required. For example, `Impala` [45] is an SQL query engine. A bug report [46] of `Impala` complains about a hang in its build process. However, programmers cannot reproduce it, and the problem disappears after restarting the build task. As a result, no repairs are applied.

R4.2 Unfixed problems (7/200, 3.5%). Programmers can choose to live with a problem of a bug report, in that it is difficult to fix it. For example, a bug report [91] of `PDFBox` [92] claims an out-of-memory crash. Programmers identify that the garbage collector of JVM does not collect caches in time, and causes this crash. They open a new task [93], but it is still unassigned.

Finding 7 *A workaround can have no repairs due to two reasons: (1) some bugs are unnecessary to be fixed (4.5%) and (2) some cannot be repaired at present (3.5%).*

5.4 RQ4. Associations

5.4.1 Protocol

In this research question, we analyze the associations of symptoms, causes and repairs, and we use the *lift* function [112] to measure the associations:

$$lift(A,B) = \frac{P(A \cap B)}{P(A) * P(B)} \tag{2}$$

where $P(A)$, $P(B)$, $P(A \cap B)$ denote the probabilities that a workaround belongs to category A, category B, and their intersection. Typically, only when the *lift* value is greater than one, category A and B are associated. To ensure the reliability of our found associations, we identify an association, only when the *lift* value is greater than 1.5 and the intersection of A and B contains at least 5 workarounds. In this research question, we explore symptom(A)-to-cause(B) and cause(A)-to-repair(B) associations.

5.4.2 Result

Figure 5 shows the associations of symptoms, causes, and repairs. In this Figure, each column denotes one of the above three dimensions, and a node of a dimension denotes a category of the dimension. An edge between two nodes denotes an association between the two corresponding categories, and a thicker edge denotes a larger *lift* value. The thickness of an edge is in proportion to its *lift* value calculated by Equation 2. The observation leads to a finding:

Finding 8 *Crashes are fixed as workarounds, often because their problems reside in programming environments. Unexpected behaviors are fixed as workarounds, often because their repairs have flaws or the cause is incompatible issues.*

We find that most associations between causes and repairs are straightforward. For example, if the operating system causes a bug, a typical workaround is to select alternative operating systems.

Lamothe and Shang [120] define four patterns to detect API workarounds. Their first pattern, functionality extensions, corresponds to our *R1.11 Overridden APIs*. Their second pattern, deep copies, corresponds to our *R1.7 Deep copying*. We do not find the correspondences for their other two patterns such as multi-versions and unnecessary workarounds. The overlapped categories are both rare in our inspected workarounds (0.90% for functionality extensions and 2.71% for deep copies). From the perspective of causes, their workarounds are caused by problems in my libraries. To fix such problems as workarounds, we find that programmers tend more to switch to other versions of libraries or change their settings. We find that the patterns of the prior study [120] are less common in real projects, because they can introduce incompatible issues. For example, if programmers modify the code [101] or deep copy the code [120] of a library, their client code becomes incompatible with the library. This type of resolutions can cause many problems.

5.5 Threats to Validity

The threats to validity are as follows:

The threats to internal validity include the possible errors in our manual classification. To reduce the threat, we split our manual analysis into two phases, and ask different authors to inspect the results in each phase. In addition, to resolve our disagreements, we contact the reporters and programmers who handle the workarounds. Furthermore, we release our inspection results on our project website, so other researchers can recheck the results.

The threats to external validity include our subjects. All workarounds are selected from Apache projects. Programmers in other projects can follow other guidelines to handle workarounds, and it can require different repairs to resolve more recent techniques. Due to the heavy effort of manual analysis, it is infeasible to analyze many subjects, and our subject size is comparable with those of the prior ones [120]. Although we cannot guarantee that we have obtained a full picture of workarounds, we have identified many types of workarounds that are less known to researchers. This threat can be further reduced if other researchers supplement more subjects.

The threats to construct validity threat include the slice of time. As our data are extracted as a snapshot, the findings in our study are valid, only for a limited period of time. With the rapid improvement of technology, future programmers could introduce other types of workarounds. This threat is shared by all empirical studies. For those important and interesting research topics, researchers can replicate empirical studies and explore to what degree their findings still hold.

6 Discussion

To resolve harmful workarounds, we interpret our findings.

Highlighting the significance of reported bugs with workarounds. Finding 3 shows that 40% of workarounds are caused by bugs in external projects, and 23.5% of workarounds are caused by programming environments. Finding 4 shows that 41% of workarounds are related to calling APIs. If the problem of a workaround resides in other projects, its programmers cannot fix it by themselves, but have to report it to the other projects. However, the programmers of the other projects often do not understand the relevance of a bug. If a tool can recommend related workarounds for a given bug report, this tool can be useful to estimate the significance of a bug report.

Identifying bugs that can be resolved as workarounds. It can be interesting to analyze the associations between workarounds and software metrics (*e.g.*, the types of bugs), which can provide actionable advice on handling workarounds. For example, Finding 1 shows that workarounds resolve more crashes than unexpected behaviors. This finding indicates that crashes can be bypassed, if they do not introduce unexpected behaviors.

Improving bug report statuses in issue trackers. According to Section 4.1 and Finding 7, several workarounds can be better marked as a different type of resolutions. The problem can be mitigated, if issue trackers define clearly defined resolutions. For example, if the problem of a bug report is fixed in new versions or related issues, it can be better marked as *fixed in other locations*. If the

problem of a bug report results in no repairs, it can be better marked as *won't fix* or *technical debts*.

7 Conclusion and Future Work

In issue trackers, workarounds are less known and more ambiguous than other bug reports. To deepen the knowledge on workarounds, we conducted an empirical study on 200 workarounds that were collected from Apache projects. In this study, we systematically analyzed the symptoms, causes, repairs of workarounds, the correlations among them, and the definitions of workarounds. Our study reveals many patterns to handle bugs as workarounds, and we present various examples to illustrate workarounds. They are useful for programmers to handle similar bugs. Furthermore, we summarized our analysis results into eight findings, and for the first time, we presented the answers to four open questions based on these findings. Based on these findings, researchers can improve the designs of APIs and issue trackers. In future work, we plan to extend our work from the following perspectives: (1) proposing techniques to detect workarounds; (2) analyzing the evolution of workarounds (*e.g.*, whether workarounds are finally removed after bugs are fully fixed); and (3) analyzing the relations between technical debt and workarounds.

Acknowledgements

We appreciate reviewers for their insightful comments. This work is sponsored by the National Nature Science Foundation of China No. 62232003 and 62272295.

Data Availability

The data of this study have been deposited in the following public repository:
`https://github.com/tetradecane/Workaround_journal_website`

References

1. `https://issues.apache.org/jira` (2020)
2. `http://www.bugzilla.org/` (2020)
3. `https://issues.apache.org/jira/browse/BEAM-6460` (2020)
4. `https://beam.apache.org/get-started/beam-overview/` (2020)
5. `https://flink.apache.org/` (2020)
6. `https://github.com/apache/beam/pull/7552` (2020)
7. `https://issues.apache.org/jira/browse/FLINK-10928` (2020)
8. `https://github.com/apache/flink/` (2020)
9. `https://github.com/tensorflow/` (2020)
10. `https://projects.apache.org/projects.html?category` (2020)
11. `http://jclouds.apache.org` (2020)
12. `https://issues.apache.org/jira/browse/ODFTOOLKIT-375` (2020)
13. `https://issues.apache.org/jira/browse/ZEPPELIN-1966` (2020)
14. `https://issues.apache.org/jira/browse/HIVEMALL-30` (2020)
15. `https://spark.apache.org/` (2020)
16. `https://issues.apache.org/jira/browse/SPOT-26` (2020)
17. `https://issues.apache.org/jira/browse/SPOT-238` (2020)

18. https://issues.apache.org/jira/browse/SOLR-11386 (2020)
19. https://issues.apache.org/jira/browse/FOP-1872 (2020)
20. https://github.com/apache/xmlgraphics-fop/commit/72d4de19de6175a92213e3c05e11b71dd1ed2714 (2020)
21. https://issues.apache.org/jira/browse/GEODE-2412 (2020)
22. https://issues.apache.org/jira/browse/XGC-66 (2020)
23. https://github.com/apache/xmlgraphics-commons/commit/3019cac528aa9de78c3a69c173bbb64642b4292a (2020)
24. https://issues.apache.org/jira/browse/SPARK-13246 (2020)
25. https://issues.apache.org/jira/browse/XERCESJ-242 (2020)
26. https://issues.apache.org/jira/browse/JS1-112 (2020)
27. https://issues.apache.org/jira/browse/RANGER-2149 (2020)
28. https://issues.apache.org/jira/browse/XERCESC-759 (2020)
29. https://issues.apache.org/jira/browse/ASTERIXDB-1767 (2020)
30. https://issues.apache.org/jira/browse/AXIS-26 (2020)
31. http://thrift.apache.org/ (2020)
32. https://hackage.haskell.org/package/hspec-core (2020)
33. https://issues.apache.org/jira/browse/THRIFT-4044 (2020)
34. https://logging.apache.org/log4j/2.x/ (2020)
35. https://issues.apache.org/jira/browse/LOG4J2-2281 (2020)
36. http://www-01.ibm.com/support/docview.wss?uid=swg1PI89708 (2020)
37. https://issues.apache.org/jira/browse/SPARK-13612 (2020)
38. https://issues.apache.org/jira/browse/SPARK-11516 (2020)
39. https://spark.apache.org/docs/latest/monitoring.html (2020)
40. http://mesos.apache.org/ (2020)
41. https://issues.apache.org/jira/browse/MESOS-7216 (2020)
42. http://openjpa.apache.org/ (2020)
43. https://issues.apache.org/jira/browse/OPENJPA-2671 (2020)
44. http://spot.incubator.apache.org/ (2020)
45. http://impala.apache.org/ (2020)
46. https://issues.apache.org/jira/browse/IMPALA-2511 (2020)
47. https://issues.apache.org/jira/browse/CB-11974 (2020)
48. https://github.com/maklesoft/cordova-plugin-migrate-localstorage (2020)
49. https://issues.apache.org/jira/browse/MESOS-5441 (2020)
50. https://issues.apache.org/jira/browse/CB-13038 (2020)
51. http://cordova.apache.org/ (2020)
52. https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries (2020)
53. https://netbeans.apache.org/ (2020)
54. https://issues.apache.org/jira/browse/NETBEANS-2840 (2020)
55. https://lucene.apache.org/solr/ (2020)
56. https://issues.apache.org/jira/browse/SOLR-8876 (2020)
57. https://issues.apache.org/jira/browse/JCLOUDS-1293 (2020)
58. https://activemq.apache.org/components/artemis/ (2020)
59. https://issues.apache.org/jira/browse/ARTEMIS-2132 (2020)
60. https://tinyurl.com/y4zjt352 (2020)
61. http://synapse.apache.org (2020)
62. https://issues.apache.org/jira/browse/SYNAPSE-1098 (2020)
63. https://issues.apache.org/jira/browse/SOLR-6209 (2020)
64. http://nifi.apache.org/registry.html (2020)
65. https://issues.apache.org/jira/browse/NIFIREG-142 (2020)
66. http://openmeetings.apache.org/ (2020)
67. https://moodle.org/ (2020)
68. https://github.com/openmeetings/openmeetings-moodle-plugin (2020)
69. https://issues.apache.org/jira/browse/OPENMEETINGS-1575 (2020)
70. http://maven.apache.org/ (2020)
71. http://fusesource.github.io/jansi (2020)
72. https://issues.apache.org/jira/browse/MNG-6417 (2020)
73. https://tinyurl.com/y3en9f3d (2020)
74. https://nemo.apache.org (2020)
75. https://issues.apache.org/jira/browse/NEMO-416 (2020)

76. https://tinyurl.com/y2v32wmk (2020)
77. https://zookeeper.apache.org/ (2020)
78. https://issues.apache.org/jira/browse/SOLR-11049 (2020)
79. https://issues.apache.org/jira/browse/SOLR-11250 (2020)
80. https://weex.apache.org/ (2020)
81. https://issues.apache.org/jira/browse/WEEX-229 (2020)
82. https://issues.apache.org/jira/browse/ZEPPELIN-4002 (2020)
83. http://tinkerpop.apache.org/ (2020)
84. https://issues.apache.org/jira/browse/TINKERPOP-2286 (2020)
85. https://issues.apache.org/jira/browse/ATLAS-1057 (2020)
86. http://atlas.apache.org/ (2020)
87. https://mina.apache.org/sshd-project/ (2020)
88. https://issues.apache.org/jira/browse/SSHD-754 (2020)
89. https://issues.apache.org/jira/browse/ZEPPELIN-3195 (2020)
90. https://issues.apache.org/jira/browse/NETBEANS-2733 (2020)
91. https://issues.apache.org/jira/browse/PDFBOX-4396 (2020)
92. http://pdfbox.apache.org/ (2020)
93. https://issues.apache.org/jira/browse/PDFBOX-4668 (2020)
94. https://github.com/NixOS/nix/issues/7815 (2023)
95. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. ICDE, pp. 3–14 (1995)
96. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proc. POPL, pp. 4–16 (2002)
97. Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? In: Proc. ICSE, pp. 361–370 (2006)
98. Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T.: What makes a good bug report? In: Proc. ESEC/FSE, pp. 308–318 (2008)
99. Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S.: Duplicate bug reports considered harmful? really? In: Proc. ICSM, pp. 337–345 (2008)
100. Bhattacharya, P., Ulanova, L., Neamtiu, I., Koduru, S.C.: An empirical analysis of bug reports and bug fixing in open source android apps. In: Proc. CSMR, pp. 133–143 (2013)
101. Bogart, C., Kästner, C., Herbsleb, J., Thung, F.: How to break an API: cost negotiation and community values in three software ecosystems. In: Proc. ESEC/FSE, pp. 109–120 (2016)
102. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: Proc. ISSTA, pp. 85–96 (2010)
103. Dehaghani, S.M.H., Hajrahimi, N.: Which factors affect software projects maintenance cost more? Acta Informatica Medica **21**(1), 63 (2013)
104. Endrikat, S., Hanenberg, S., Robbes, R., Stefik, A.: How do API documentation and static typing affect API usability? In: Proc. ICSE, pp. 632–642 (2014)
105. Engler, D.R., Chen, D.Y., Chou, A.: Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In: Proc. SOSP, pp. 57–72 (2001)
106. Erlikh, L.: Leveraging legacy system dollars for e-business. IT professional **2**(3), 17–23 (2000)
107. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1-3), 35–45 (2007)
108. Francalanci, C., Merlo, F.: Empirical analysis of the bug fixing process in open source projects. In: Proc. OSS, pp. 187–196 (2008)
109. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: Proc. ESEC/FSE, pp. 339–349 (2008)
110. Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B.: Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Proc. ICSE, pp. 495–504 (2010)
111. Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B.: Not my bug! and other reasons for software bug report reassignments. In: Proc. CSCW, pp. 395–404 (2011)
112. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers (2011)
113. Herzig, K., Just, S., Zeller, A.: It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proc. ICSE, pp. 392–401 (2013)
114. Hora, A.C., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., Valente, M.T.: How do developers react to API evolution? the pharo ecosystem case. In: Proc. ICSME, pp. 251–260 (2015)

115. Jeong, G., Kim, S., Zimmermann, T.: Improving bug triage with bug tossing graphs. In: Proc. ESEC/FSE, p. 111–120 (2009)
116. Jia, L., Zhong, H., Wang, X., Huang, L., Lu, X.: An empirical study on bugs inside tensorflows. In: Proc. DASFAA, p. to appear (2020)
117. Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., Shapochka, A.: A case study in locating the architectural roots of technical debt. In: Proc. ICSE, vol. 2, pp. 179–188 (2015)
118. Krippendorff, K.: Computing Krippendorff's alpha-reliability (2011)
119. Lamothe, M., Guéhéneuc, Y.G., Shang, W.: A systematic review of api evolution literature. ACM Computing Surveys **54**(8), 1–36 (2021)
120. Lamothe, M., Shang, W.: When APIs are intentionally bypassed: An exploratory study of API workarounds. In: Proc. ICSE, p. to appear (2020)
121. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE transactions on software engineering **38**(1), 54–72 (2011)
122. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C.: Have things changed now? an empirical study of bug characteristics in modern open source software. In: Proc. ASID, pp. 25–33 (2006)
123. Li, Z., Zhong, H.: An empirical study on obsolete issue reports. In: Proc. ASE, p. to eappear (2021)
124. Li, Z., Zhou, Y.: Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. ESEC/FSE, pp. 306–315 (2005)
125. Lin, Z., Shu, F., Yang, Y., Hu, C., Wang, Q.: An empirical study on bug assignment automation using chinese bug data. In: Proc. ESEM, pp. 451–455 (2009)
126. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proc. ICSE, pp. 501–510 (2008)
127. Monperrus, M., Eichberg, M., Tekes, E., Mezini, M.: What should developers be aware of? an empirical study on the directives of API documentation. Empirical Software Engineering **17**(6), 703–737 (2012)
128. Nadi, S., Krüger, S., Mezini, M., Bodden, E.: Jumping through hoops: why do java developers struggle with cryptography apis? In: Proc. ICSE, pp. 935–946 (2016)
129. Nawaz, A.: A comparison of card-sorting analysis methods. In: APCHI, pp. 28–31 (2012)
130. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Graph-based mining of multiple object usage patterns. In: Proc. ESEC/FSE, pp. 383–392 (2009)
131. Okur, S., Dig, D.: How do developers use parallel libraries? In: Proc. ESEC/FSE, p. 54 (2012)
132. Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S., Paradkar, A.M.: Inferring method specifications from natural language API descriptions. In: Proc. ICSE, pp. 815–825 (2012)
133. Potdar, A., Shihab, E.: An exploratory study on self-admitted technical debt. In: Proc. ICSME, pp. 91–100 (2014)
134. Qiu, D., Li, B., Leung, H.: Understanding the API usage in java. Information and Software Technology **73**, 81–100 (2016)
135. Ramasubbu, N., Kemerer, C.F.: Integrating technical debt management and software quality management processes: A normative framework and field tests. IEEE Transactions on Software Engineering **45**(3), 285–300 (2019)
136. Robbes, R., Lungu, M., Röthlisberger, D.: How do developers react to API deprecation?: the case of a smalltalk ecosystem. In: Proc. ESEC/FSE, p. 56 (2012)
137. Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T.: Automated API property inference techniques. IEEE Transactions on Software Engineering **39**(5), 613–637 (2013)
138. Robillard, M.P., DeLine, R.: A field study of API learning obstacles. Empirical Software Engineering **16**(6), 703–732 (2011)
139. Seacord, R.C., Plakosh, D., Lewis, G.A.: Modernizing legacy systems: software technologies, engineering processes, and business practices. Addison-Wesley Professional (2003)
140. Shi, L., Zhong, H., Xie, T., Li, M.: An empirical study on evolution of API documentation. In: Proc. ETAPS/FASE, pp. 416–431 (2011)
141. Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C.: Bug characteristics in open source software. Empirical Software Engineering **19**(6), 1665–1705 (2014)
142. Tang, Y., Khatchadourian, R., Bagherzadeh, M., Singh, R., Stewart, A., Raja, A.: An empirical study of refactorings and technical debt in machine learning systems. In: Proc. ICSE, pp. 238–250 (2021)

143. Thung, F., Wang, S., Lo, D., Jiang, L.: An empirical study of bugs in machine learning systems. In: Proc. ISSRE, pp. 271–280 (2012)
144. Tom, E., Aurum, A., Vidgen, R.: An exploration of technical debt. Journal of Systems and Software **86**(6), 1498–1516 (2013)
145. Vásquez, M.L., Bavota, G., Bernal-Cárdenas, C., Penta, M.D., Oliveto, R., Poshyvanyk, D.: API change and fault proneness: a threat to the success of android apps. In: Proc. ESEC/FSE, pp. 477–487 (2013)
146. Vásquez, M.L., Bavota, G., Penta, M.D., Oliveto, R., Poshyvanyk, D.: How do API changes trigger stack overflow discussions? a study on the android SDK. In: Proc. ICPC, pp. 83–94 (2014)
147. Vetrò, A.: Using automatic static analysis to identify technical debt. In: Proc. ICSE, pp. 1613–1615 (2012). DOI 10.1109/ICSE.2012.6227226
148. Xia, X., Lo, D., Wen, M., Shihab, E., Zhou, B.: An empirical study of bug report field reassignment. In: Proc. CSMR-WCRE, pp. 174–183 (2014)
149. Yan, A., Zhong, H., Song, D., Jia, L.: The symptoms, causes, and repairs of workarounds in apache issue trackers. In: Proc. ICSE (2022)
150. Yan, M., Xia, X., Shihab, E., Lo, D., Yin, J., Yang, X.: Automating change-level self-admitted technical debt determination. IEEE Transactions on Software Engineering **45**(12), 1211–1229 (2018)
151. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Elsevier (2006)
152. Zhang, F., Khomh, F., Zou, Y., Hassan, A.E.: An empirical study on factors impacting bug fixing time. In: Proc. WCRE, pp. 225–234 (2012)
153. Zhang, H., Gong, L., Versteeg, S.: Predicting bug-fixing time: an empirical study of commercial software projects. In: Proc. ICSE, pp. 1042–1051 (2013)
154. Zhang, Y., Chen, Y., Cheung, S., Xiong, Y., Zhang, L.: An empirical study on TensorFlow program bugs. In: Proc. ISSTA, pp. 129–140 (2018)
155. Zhong, H.: Enriching compiler testing with real program from bug report. In: Proc. ASE, pp. 1–12 (2022)
156. Zhong, H., Mei, H.: An empirical study on API usages. IEEE Transaction on software engineering **45**, 319–334 (2018)
157. Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., Weiss, C.: What makes a good bug report? IEEE Transactions on Software Engineering **36**(5), 618–643 (2010)