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

# Demystifying the Challenges and Benefits of Analyzing User-Reported Logs in Bug Reports

**An Ran Chen · Tse-Hsun (Peter) Chen · Shaowei Wang**

Received: date / Accepted: date

**Abstract** Logs in bug reports provide important debugging information for developers. During the debugging process, developers need to study the bug report and examine user-provided logs to understand the system executions that lead to the problem. Intuitively, user-provided logs illustrate the problems that users encounter and may help developers with the debugging process. However, some logs may be incomplete or inaccurate, which can cause difficulty for developers to diagnose the bug, and thus, delay the bug fixing process. In this paper, we conduct an empirical study on the challenges that developers may encounter when analyzing the user-provided logs and their benefits. In particular, we study both log snippets and exception stack traces in bug reports. We conduct our study on 10 large-scale open-source systems with a total of 1,561 bug reports with logs (BRWL) and 7,287 bug reports without logs (BRNL). Our findings show that: 1) BRWL takes longer time (median ranges from 3 to 91 days) to resolve compared to BRNL (median ranges from 1 to 25 days). We also find that reporters may not attach accurate or sufficient logs (i.e., developers often ask for additional logs in the Comments section of a bug report), which extends the bug resolution time. 2) Logs often provide a good indication of where a bug is located. Most bug reports (73%) have overlaps between the classes that generate the logs and their corresponding fixed classes. However, there is still a large number of bug reports where there is no overlap between the logged and fixed classes. 3) Our manual study finds that there is often missing system execution information in the logs. Many logs only show the point of failure (e.g., exception) and do not provide a direct hint on the actual root cause. In fact, through call graph analysis, we find that 28%

An Ran Chen · Tse-Hsun (Peter) Chen
Software PErformance, Analysis, and Reliability (SPEAR) Lab, Concordia University, Montreal, Canada
Shaowei Wang
Department of Computer Science, University of Manitoba, Manitoba, Canada
Email: {anr_chen, peterc}@encs.concordia.ca, shaowei@cs.umanitoba.ca

of the studied bug reports have the fixed classes reachable from the logged classes, while are not visible in the logs attached in bug reports. In addition, some logging statements are removed in the source code as the system evolves, which may cause further challenges in analyzing the logs. In short, our findings highlight possible future research directions to better help practitioners attach or analyze logs in bug reports.

**Keywords** bug report, log, stack trace, empirical study.

## 1 Introduction

Software debugging is an important and challenging task in software maintenance. As the complexity of modern software systems increases, developers need to spend more time on understanding system execution in order to locate the problem. A prior study (LaToza and Myers, 2010) finds that developers, on average, spend 33% of their time on debugging. To assist developers with debugging, prior studies (Bianchi *et al.*, 2017; Hassani *et al.*, 2018; Jin and Orso, 2012; Li *et al.*, 2020a; Soltani *et al.*, 2018; Wu *et al.*, 2014; Yuan *et al.*, 2010, 2011) propose approaches to analyze crash reports or system logs. However, these prior approaches often assume that developers have access to the entire system-generated logs or instrumented system runtime data. In practice, such information may not always be available due to privacy or technical concerns (Cao *et al.*, 2014; Satvat and Saxena, 2018; Shang *et al.*, 2013). For instance, users usually only attach a portion of the logs in their bug reports, since the size of the entire log file is often several gigabytes or even larger (Chen *et al.*, 2017; Shang *et al.*, 2013).

Bug reports provide important information for developers to fix the problems that users encounter (Anvik *et al.*, 2006; Bettenburg *et al.*, 2008b). Typically, when reporters create a bug report, they need to provide a title, the severity (e.g., major or minor), the description of the problem, and system-generated logs (e.g., log messages or stack traces) which illustrate the system execution paths when the problem occurs. In particular, such logs may contain valuable debugging information for developers (Bettenburg *et al.*, 2008b; Schroter *et al.*, 2010; Zimmermann *et al.*, 2010). Based on the user-provided information, developers then diagnose the problem and resolve the issue. In general, developers first look at the description of the bug report and manually examine the attached logs. Then, developers investigate where the logs were generated in the source code to find out where the bug might be. Finally, developers manually examine the source code and the corresponding logs, trying to understand how the system was executed when the bug happened and resolve the bug.

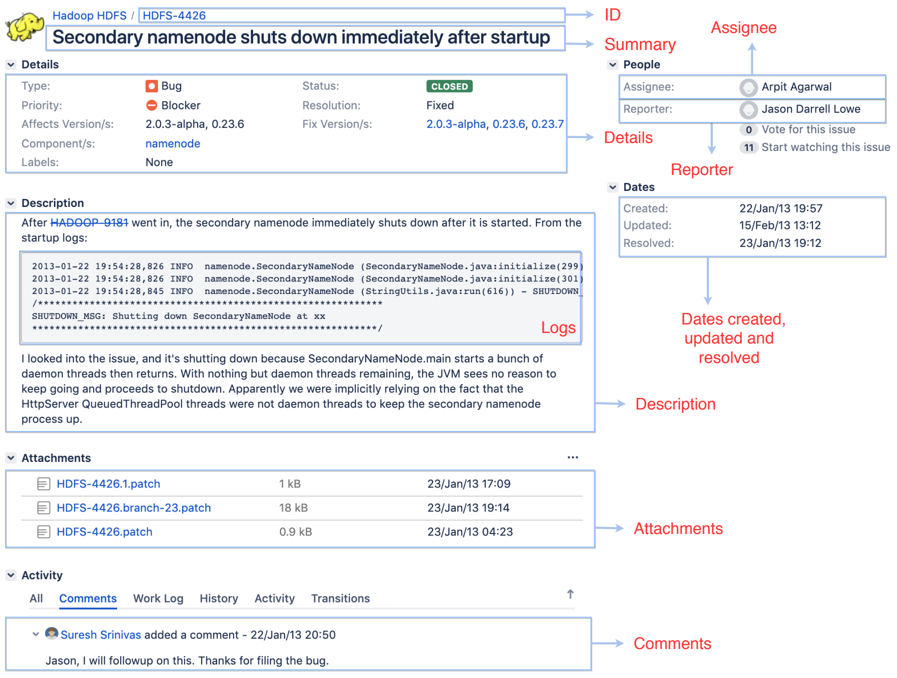
Intuitively, user-provided logs in bug reports illustrate the problems that users encounter and may help developers with the debugging process (Bettenburg *et al.*, 2008b). However, some logs may be incomplete or inaccurate, which can cause difficulty for developers to diagnose the bug, and thus, delay the bug fixing process. In this paper, we study the usefulness of logs in bug reports and the challenges that developers may encounter when analyzing such logs. We conduct our study on 10 open-source systems (i.e., ActiveMQ, AspectJ, Hadoop Common, HDFS, MapReduce, YARN, Hive, PDE, Storm, and Zookeeper), which are commonly used in prior log-related studies (Chen and Jiang, 2017; Li *et al.*, 2020a, 2019; Yuan *et al.*, 2014). In particular, we seek to answer the following research questions:

– **RQ1) Are bug reports with logs resolved faster than bug reports without logs?**
  Different from prior studies, our results suggest that bug reports with logs take longer time to resolve (median ranges from 3 to 91 days) than those without logs (median ranges from 1 to 25 days). Our further analysis shows that developers often ask for more logs in the Comments section of a bug report, which extends bug resolution time.
– **RQ2) Are there overlaps between logged classes and fixed classes?**
  We find that 73% (995/1,370) of the bug reports with logs have overlaps between the logged classes and fixed classes. Although the logged classes can locate up to 51.6% (44% on average) of the fixed classes, there is still an average of 56% of the fixed classes that have no overlap with the logged classes.
– **RQ3) Why do some fixed classes have no overlap with the logged classes?**
  We conduct a manual study on the bug reports where there is no overlap between the logged classes and fixed classes. We find that most logs only record the unexpected behavior of the system (e.g., exception) but do not show the root cause of a bug nor the execution that led to the failure. We also find that some logging statements are removed during code evolution, so the logs can no longer be mapped to the source code.

In summary, our findings show the benefits of user-reported logs in debugging bug reports and potential challenges. Future studies should assist reporters to attach logs that can more accurately show the execution that lead to the root cause of a bug. In addition, our manual study finds that approaches that can help developers recover the system execution by connecting the logs may be also helpful. To facilitate the reproducibility, we have made the data available online[1].

**Paper Organization.** Section 2 provides an overview on the background. Section 3 explains our case study setup. Section 4 answers our research questions. Section 5 summarizes the implications of our findings. Section 6 discusses the threats to validity. Section 7 surveys related work. Finally, Section 8 concludes this paper.

[1] https://github.com/SPEAR-SE/LogInBugReportsEmpirical_Data



Hadoop HDFS / HDFS-4426

Secondary namenode shuts down immediately after startup

Details

| | | | |
|---|---|---|---|
| Type: | Bug | Status: | CLOSED |
| Priority: | Blocker | Resolution: | Fixed |
| Affects Version/s: | 2.0.3-alpha, 0.23.6 | Fix Version/s: | 2.0.3-alpha, 0.23.6, 0.23.7 |
| Component/s: | namenode | | |
| Labels: | None | | |

People

| | |
|---|---|
| Assignee: | Arpit Agarwal |
| Reporter: | Jason Darrell Lowe |

0 Vote for this issue

11 Start watching this issue

Dates

| | |
|---|---|
| Created: | 22/Jan/13 19:57 |
| Updated: | 15/Feb/13 13:12 |
| Resolved: | 23/Jan/13 19:12 |

Description

After ~~HADOOP-9181~~ went in, the secondary namenode immediately shuts down after it is started. From the startup logs:

```
2013-01-22 19:54:28,826 INFO  namenode.SecondaryNameNode (SecondaryNameNode.java:initialize(299)
2013-01-22 19:54:28,826 INFO  namenode.SecondaryNameNode (SecondaryNameNode.java:initialize(301)
2013-01-22 19:54:28,845 INFO  namenode.SecondaryNameNode (StringUtils.java:run(616)) - SHUTDOWN_
/************************************************************
SHUTDOWN_MSG: Shutting down SecondaryNameNode at xx
************************************************************/
```

I looked into the issue, and it's shutting down because SecondaryNameNode.main starts a bunch of daemon threads then returns. With nothing but daemon threads remaining, the JVM sees no reason to keep going and proceeds to shutdown. Apparently we were implicitly relying on the fact that the HttpServer QueuedThreadPool threads were not daemon threads to keep the secondary namenode process up.

Attachments

| | | |
|---|---|---|
| HDFS-4426.1.patch | 1 kB | 23/Jan/13 17:09 |
| HDFS-4426.branch-23.patch | 18 kB | 23/Jan/13 19:14 |
| HDFS-4426.patch | 0.9 kB | 23/Jan/13 04:23 |

Activity

All Comments Work Log History Activity Transitions

Suresh Srinivas added a comment - 22/Jan/13 20:50

Jason, I will followup on this. Thanks for filing the bug.

Fig. 1: An example bug report **(HADOOP-4426)** on Jira.

## 2 Background

In this section, we give a brief overview of the types of information that is available in a bug report.

### 2.1 Bug Reports

Bug reports contain information to help developers diagnose reported bugs. A prior study (Bettenburg *et al.*, 2008b) points out that from the developers' perspective, a good bug report should have a clear description and other important debugging information, such as logs. On bug tracking systems such as Jira, bug reports typically contain the following fields: Summary, Status, Details (including Type, Status, Priority, Resolution, Affects Versions, and Fix Versions), Assignee, Reporter, Description, Attachments, and Comments. Figure 1 shows an example of a bug report from the Hadoop Common system. The Summary section gives an overview of the bug. The Description section provides an explanation to the bug, and may contain the logs for debugging hints and some user-specific runtime information (e.g, describe the specific use case or hardware environment). The Status field provides the status of the bug report in the workflow. The Resolution field indicates the final resolution assigned to the reported bug (e.g., FIXED, DUPLICATE, WON'T FIX).

```
2009-02-12 08:35:36,417 INFO org.apache.hadoop.mapred.TaskTracker:
    Task attempt_200902120746_0297_r_000033_0 is in COMMIT_PENDING
2009-02-12 08:35:36,417 INFO org.apache.hadoop.mapred.TaskTracker:
    attempt_200902120746_0297_r_000033_0 0.33333334% reduce > sort
```

Fig. 2: An example of log snippets. (**HADOOP-5233**)

```
17/07/14 13:31:58 INFO hdfs.DFSClient: Exception in createBlock
-OutputStream java.io.EOFException:
  at org.apache.hadoop.hdfs.protocolPB.PBHelper.vintPrefixed(PB
  -Helper.java:2280)
  at org.apache.hadoop.hdfs.DFSOutputStream$DataStreamer.create
  -BlockOutputStream(DFSOutputStream.java:1318)
  ...
```

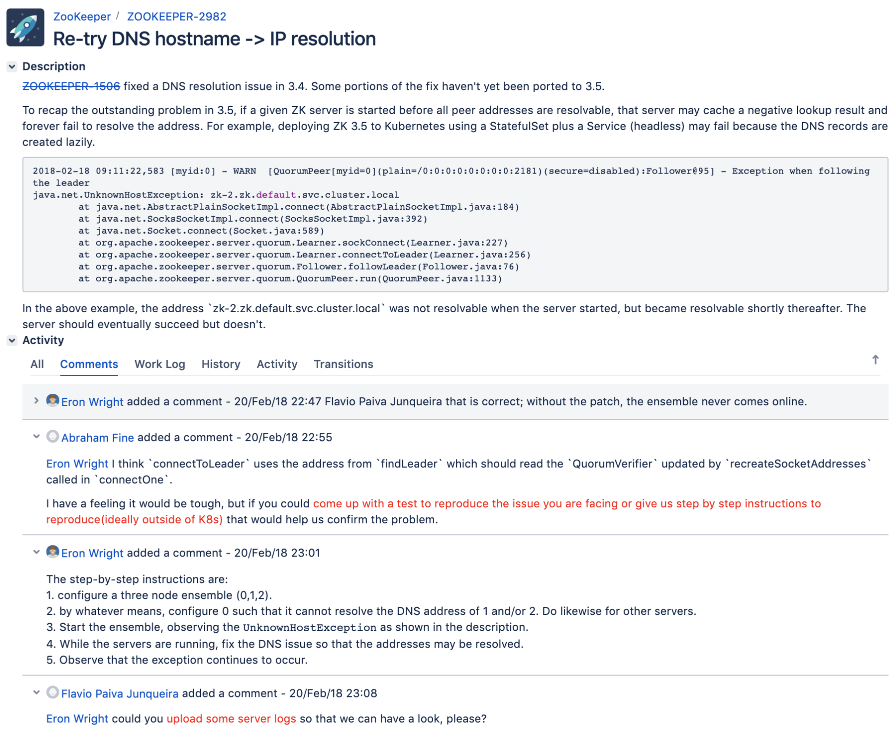
Fig. 3: An example of exception logs. (**HDFS-8475**)

The Affects Versions field is usually provided by the reporter, whereas the Fix Versions field is added by the assignee after bug fixes. Sometimes, either the reporter or the assignee might attach patches or tests in the Attachments section. In the Comments section, developers may further discuss the bug, provide opinions, and potentially ask for additional technical details. To note that the same idea applies for bug reports on Bugzilla, although they do not contain the Fix Versions field.

## 2.2 Logs in Bug Reports

To assist developers to diagnose and fix bugs, reporters may attach logs in their bug reports. Typically, there are two types of logs in bug reports: **log snippets**, which record software system execution at run time; and **exception logs**, which record the stack traces when an exception happens. Figure 2 and Figure 3 show an example of log snippets and exception logs, respectively. A log snippet is an ordered set of log messages generated by logging statements during runtime. Each log is often composed of the timestamp, verbosity level (e.g., debug, info, error, or fatal), class name, and detailed log message. An exception log contains information on multiple sets of stack frames (i.e., stack trace) when an exception happens. Exception logs are recorded together with log snippets to provide a more detailed view of the system execution when an exception happens (Fu *et al.*, 2014). Exception logs often contain the timestamps, thrown exceptions (e.g., NullPointerException), and the fully-qualified file names, method signatures, and line numbers for the method calls on the stack frames. In this study, we refer to *logged classes* as the classes that generate the logs, either log snippets, exception logs, or both.

In bug reports, logs may be attached in the Description and Comments sections. Reporters often open a bug report and report the failing stack traces



ZooKeeper / ZOOKEEPER-2982

Re-try DNS hostname -> IP resolution

Description

~~ZOOKEEPER-1506~~ fixed a DNS resolution issue in 3.4. Some portions of the fix haven't yet been ported to 3.5.

To recap the outstanding problem in 3.5, if a given ZK server is started before all peer addresses are resolvable, that server may cache a negative lookup result and forever fail to resolve the address. For example, deploying ZK 3.5 to Kubernetes using a StatefulSet plus a Service (headless) may fail because the DNS records are created lazily.

```
2018-02-18 09:11:22,583 [myid:0] - WARN  [QuorumPeer[myid=0](plain=/0:0:0:0:0:0:0:0:2181)(secure=disabled):Follower@95] - Exception when following
the leader
java.net.UnknownHostException: zk-2.zk.default.svc.cluster.local
        at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:184)
        at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
        at java.net.Socket.connect(Socket.java:589)
        at org.apache.zookeeper.server.quorum.Learner.sockConnect(Learner.java:227)
        at org.apache.zookeeper.server.quorum.Learner.connectToLeader(Learner.java:256)
        at org.apache.zookeeper.server.quorum.Follower.followLeader(Follower.java:76)
        at org.apache.zookeeper.server.quorum.QuorumPeer.run(QuorumPeer.java:1133)
```

In the above example, the address `zk-2.zk.default.svc.cluster.local` was not resolvable when the server started, but became resolvable shortly thereafter. The server should eventually succeed but doesn't.

Activity

Eron Wright added a comment - 20/Feb/18 22:47 Flavio Paiva Junqueira that is correct; without the patch, the ensemble never comes online.

Abraham Fine added a comment - 20/Feb/18 22:55

Eron Wright I think `connectToLeader` uses the address from `findLeader` which should read the `QuorumVerifier` updated by `recreateSocketAddresses` called in `connectOne`.

I have a feeling it would be tough, but if you could come up with a test to reproduce the issue you are facing or give us step by step instructions to reproduce(ideally outside of K8s) that would help us confirm the problem.

Eron Wright added a comment - 20/Feb/18 23:01

The step-by-step instructions are:
1. configure a three node ensemble (0,1,2).
2. by whatever means, configure 0 such that it cannot resolve the DNS address of 1 and/or 2. Do likewise for other servers.
3. Start the ensemble, observing the `UnknownHostException` as shown in the description.
4. While the servers are running, fix the DNS issue so that the addresses may be resolved.
5. Observe that the exception continues to occur.

Flavio Paiva Junqueira added a comment - 20/Feb/18 23:08

Eron Wright could you upload some server logs so that we can have a look, please?

Fig. 4: An example of a bug report **(ZOOKEEPER-2982)** highlighting the discussions between the reporter and developer (assignee). The bug report addresses a server problem when resolving for the host address on Zookeeper clusters. In the Comments section, the developer asked the reporter to provide some server logs to help the bug fix (higlighted in red).

or log snippets in the Description to assist developers in bug fixing. Occasionally, developers may discuss the bug report and request more logs from the reporter in the Comments section. Figure 4 shows such an example, where the developer first asked for a step-by-step instruction to reproduce the bug, then demanded server logs.

A number of prior studies aim to debug or reproduce bugs using system execution information (Jin and Orso, 2012; Soltani *et al.*, 2018; Wu *et al.*, 2014; Yuan *et al.*, 2010, 2011). However, these prior approaches often assume that developers have access to the entire system-generated logs or instrumented system runtime data. Such debugging data may not always be available to developers. In many cases, developers need to rely on data in bug reports for debugging, which may be incomplete or inaccurate (Bettenburg *et al.*, 2008b). Without helpful debugging data, it is difficult for developers to fully understand the bug and thus, quite often delay the bug fixes (Bettenburg *et al.*, 2008b). Thus, in this paper, we intend to explore whether user-provided logs provide valuable debugging hints to developers. Our findings provide an initial insight on leveraging readily-available information in bug reports to assist developers with debugging, and provide a deeper understanding of the reasons and potential solutions to the challenges that developers may encounter when analyzing user-provided logs in bug reports.

## 3 Data Collection and Case Study Setup

In this section, we first discuss the studied systems. Then, we describe our data collection process and data characteristics.

### 3.1 Studied Systems.

Table 1 shows an overview of the studied systems. We conduct our case study on 10 Java-based open source systems: ActiveMQ, AspectJ, Hadoop Common, HDFS, MapReduce, YARN, Hive, PDE, Storm and ZooKeeper. The size of the studied systems ranges from 144K to 1.7M lines of code. These studied systems are widely used in prior log-related studies and have high-quality logs (Chen and Jiang, 2017; Li *et al.*, 2019; Yuan *et al.*, 2014). The studied systems also cover different domains, varying from virtual machine deployment systems to data warehousing solutions. Most of the systems have more than 10 years of code development. We choose these systems because they are large in scale, actively maintained, well-documented, and have many bug reports that contain logs (Chen and Jiang, 2017; Li *et al.*, 2019).

### 3.2 Collecting and Filtering Bug Reports.

We collect all the bug report data that is available on the Jira repository (Apache, 2019) of each studied system from 2008 (or the earliest bug creation date) to January 2019, and compute the lines of code (LOC) on the master branch (data collected in January 2019). To collect the bug reports, we built a web crawler that sends REST API calls to the Jira repositories. We select the bug reports based on the criteria that are used in prior bug report studies (Chen and Jiang, 2017; Chen *et al.*, 2014; Yuan *et al.*, 2014). Namely, we select bug reports of the type *"Bug"*, whose status are *"Closed"* or *"Resolved"*, with the resolution *"Fixed"* and priority marked as *"Major"* or above. Additionally, we only select the bug reports that have corresponding code changes in the code repository (i.e., having commit messages that contain the bug report ID), so we can verify that the bugs are indeed fixed. At the end of this process, we collected a total of 8,848 bug reports.

Table 1: An overview of the studied systems.

| System | LOC | Type | Code maturity | Selected bug history range |
|---|---|---|---|---|
| **ActiveMQ** | 480k | Messaging server | > 10 years | 2008-01-01 to 2019-01-19 |
| **AspectJ** | 447k | Aspect-oriented extension | > 10 years | 2008-01-01 to 2019-01-19 |
| **Hadoop Common** | 364K | Common utilities | > 10 years | 2008-01-01 to 2019-01-19 |
| **HDFS** | 560K | Distributed storage | > 10 years | 2008-01-01 to 2019-01-19 |
| **MapReduce** | 291K | Distributed processing system | > 10 years | 2008-01-01 to 2019-01-19 |
| **YARN** | 313K | Resource manager | > 5 years | 2012-07-18 to 2019-01-19 |
| **Hive** | 1.7M | Data warehouse | > 5 years | 2008-10-15 to 2019-01-19 |
| **PDE** | 369k | Tools for plug-ins development | > 10 years | 2008-01-01 to 2019-01-19 |
| **Storm** | 346k | Distributed processing system | > 5 years | 2013-12-11 to 2019-01-19 |
| **Zookeeper** | 144k | Configuration service | > 10 years | 2008-06-10 to 2019-01-19 |
| **Total** | 5.0M | - | - | - |

Table 2: Bug reports in the studied systems. *BR* represents bug reports; *BRNL* represents bug reports with no logs and *BRWL* represents bug reports with logs (i.e., either contain log snippets, stack traces, or both).

| System | BR with only log snippets | BR with only stack traces | BR with both | Total BRWL | Total BRNL | Total BR |
|---|---|---|---|---|---|---|
| **ActiveMQ** | 10 | 55 | 27 | 92 (15%) | 502 (85%) | 594 |
| **AspectJ** | 0 | 42 | 3 | 45 (24%) | 140 (76%) | 185 |
| **Hadoop Common** | 23 | 71 | 58 | 152 (21%) | 573 (79%) | 725 |
| **HDFS** | 29 | 99 | 74 | 202 (17%) | 964 (83%) | 1,166 |
| **MapReduce** | 27 | 100 | 66 | 193 (34%) | 382 (66%) | 575 |
| **YARN** | 29 | 147 | 96 | 272 (47%) | 304 (53%) | 576 |
| **Hive** | 4 | 109 | 16 | 129 (6%) | 2,102 (94%) | 2,231 |
| **PDE** | 23 | 342 | 0 | 365 (17%) | 1,763 (83%) | 2,128 |
| **Storm** | 7 | 44 | 13 | 64 (17%) | 316 (83%) | 380 |
| **Zookeeper** | 9 | 20 | 18 | 47 (16%) | 241 (84%) | 288 |
| **Total** | 161 | 1,029 | 371 | 1,561 (18%) | 7,287 (82%) | 8,848 |

## 3.3 Identifying Bug Reports that Contain Logs

In this paper, **we consider two types of logs: *log snippets* and *stack traces***. We refer *log snippets* as the system-generated logs and refer *stack traces* as the reported messages in stack frames (i.e., in the case of exception). These two types of logs are often the only information that is available for debugging production problems (Fu *et al.*, 2014; Yuan *et al.*, 2010, 2012a). A log snippet is composed of consecutive log messages generated at runtime. Log messages often contain a static message (e.g., in Java, in the follow code, `Logger.info("static_message" + method())`, *static_message* is an example of a static message), values for dynamic variables, and the log verbosity level (e.g., info, warning, or error). An example log message is: *"2018-08-29 15:37:47.891 Utils [INFO] Interrupted while waiting for fencing command: cd"*, where it shows the timestamp of when the event happened, the executed class (i.e., *Utils*), the log level (i.e., *INFO*), and the log message (i.e., *Interrupted while waiting for fencing command: cd*). Note that such log messages usually contain system execution information and may not always be an indication of an error (Chen *et al.*, 2017; Yuan *et al.*, 2010). The second type of logs is the system generated exception message and stack trace. Stack traces show the stack frame of the system when exceptions occur. Typically, reporters attach logs in the bug description or as comments.

Since the studied systems use specific logging conventions on the structure of the log snippets (e.g., ordered as timestamps, verbosity level, class name, and message), we use regular expressions to capture them in the Description and Comments sections of bug reports (Chen and Jiang, 2017). Specifically, we look for log snippets by extracting lines that contain timestamps and log-related keywords (e.g., *info*, *debug*, and *error*). We look for stack traces in a similar fashion by using both keywords (e.g., a line beginning with *"at..."*) and line formats (e.g., followed by method invocation, class name, and line number) that are specific to stack traces.

## 3.4 Collected Bug Reports

***In general, we find that there is a non-negligible percentage (an average of 21.5% across all systems) of bug reports that contain logs (i.e., either log snippets, stack traces, or both).*** Table 2 shows the number of bug reports in the studied systems. We call *bug reports with logs as BRWL*, and *bug reports without logs as BRNL*. In total, 1,561 (18%) bug reports contain logs and 7,287 (82%) bug reports do not contain any logs. We also observe that 6% to 47% (an average of 21.5%) of the bug reports contain at least one type of logs, which indicates that logs are often attached by reporters to help describe problems. In addition, reporters are more likely to include stack traces in a bug report compared to log snippets. Specifically, 10% (161/1,561) of BRWL have only log snippets compared to 66% (1,029/1,561) of BRWL that have only stack traces. One possible reason is that stack traces are more straightforward to interpret (e.g., with clear exception messages and stack traces); whereas the information in the log snippets may vary depending on how reporters attach the logs and how developers write the logging statements in the source code (Li *et al.*, 2019; Yuan *et al.*, 2010, 2011). However, many bug reports still contain both log snippets and stack traces, which shows that both types of logs are commonly provided in bug reports to help debugging.

## 4 Case Study Results

In this section, we discuss the results of our research questions (RQs). For each RQ, we present the motivation, our approach and the results.

### 4.1 RQ1: Are Bug Reports With Logs Resolved Faster Than Bug Reports Without Logs?

**Motivation:** Prior studies (Bettenburg *et al.*, 2008b; Zimmermann *et al.*, 2010) found that log snippets and stack traces are useful debugging information in bug reports. Presumably, and as found in prior research (Bettenburg *et al.*, 2008b; Yuan *et al.*, 2012b; Zimmermann *et al.*, 2010), bug reports that contain logs may take a shorter amount of time to resolve compared to bug reports that do not have logs. However, prior research only studies bug reports with either log snippets or stack traces but did not study the combination of both types of logs. In addition, as also shown in Section 2, developers may ask for more logs and may thus delay the bug resolution time. Therefore, in this RQ, we revisit whether bug reports with logs are resolved faster than bug reports without logs, and if bug reports with logs in the Comments section take more time to resolve.

**Approach:** We analyze the bug resolution time for the bug reports that we collected in Section 3. In particular, we study the bug reports that have a corresponding code change in the code repository. For each analyzed bug report, we calculate the bug resolution time (in days) by taking the difference between the bug resolution date and bug report creation date (Chen *et al.*, 2014). We statistically compare the bug resolution time of the bug reports with logs (BRWL) and the bug reports without logs (BRNL). We use Wilcoxon rank-sum test to study if there exists a statistically significant difference between the resolution time of BRWL and BRNL. We select Wilcoxon rank-sum test because it is a non-parametric test that does not have an assumption on the distribution of the data (Moore *et al.*, 2009). To further show the magnitude of the difference, we compute the effect size. We use Cliff's Delta, which is also a non-parametric test, as the effect size measurement to quantify the amount of difference between BRWL and BRNL (Cliff, 1993). We assess the magnitude by using the thresholds provided by Romano *et al.* (2006):

$$\text{effect size}\begin{cases}\text{negligible,} & \text{if } |d| < 0.147\\ \text{small,} & \text{if } 0.147 \leq |d| < 0.33\\ \text{medium,} & \text{if } 0.33 \leq |d| < 0.474\\ \text{large,} & \text{if } 0.474 \leq |d|\end{cases} \tag{1}$$

**Results:** ***In general, BRWL takes more time to resolve compared to BRNL.*** Table 3 shows the median resolution time of bug reports with logs (BRWL) and without logs (BRNL). We find that the median resolution time

Table 3: A comparison of the bug resolution time (in days) between the bug reports with logs (**BRWL**) and the bug reports without logs (**BRNL**) across the studied systems.

| **Project** | **BRWL median resolution** | **BRNL median resolution** | **p-values** | **Cliff's Delta** |
|---|---|---|---|---|
| **ActiveMQ** | 21.5 | 1.0 | <0.001 | **0.73 (large)** |
| **AspectJ** | 14.0 | 25.0 | 0.89 | 0.01 (negligible) |
| **Hadoop Common** | 7.0 | 1.0 | <0.001 | **0.58 (large)** |
| **HDFS** | 27.5 | 4.0 | <0.05 | **0.46 (medium)** |
| **MapReduce** | 23.5 | 1.0 | 0.19 | 0.69 (large) |
| **YARN** | 10.0 | 2.0 | 0.47 | 0.53 (large) |
| **Hive** | 7.0 | 3.0 | 0.56 | 0.25 (small) |
| **PDE** | 3.0 | 6.0 | <0.05 | 0.11 (negligible) |
| **Storm** | 4.0 | 3.0 | <0.001 | **0.28 (small)** |
| **Zookeeper** | 91.0 | 1.0 | 0.15 | 0.88 (large) |
| **Average** | 20.9 | 4.7 | - | - |

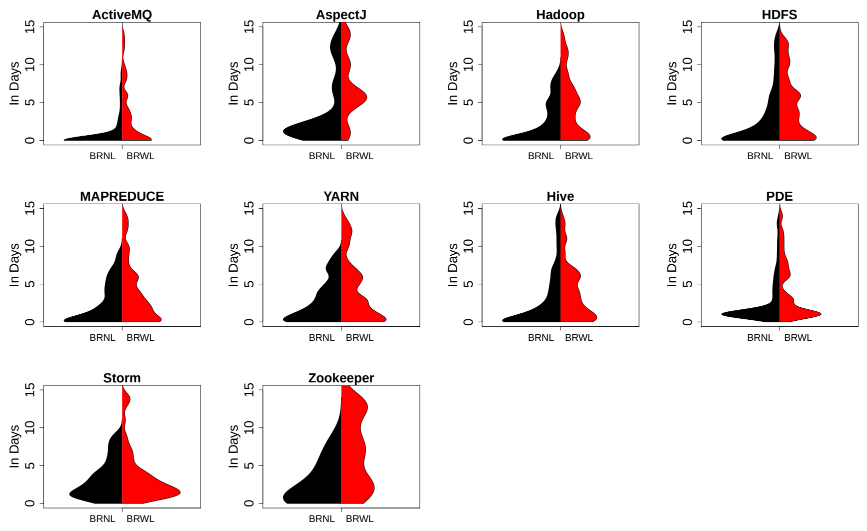


Fig. 5: Beanplots to illustrate the densities of resolution time (in days) distribution for BRWL and BRNL in range of 15 days.

ranges from 3 to 91 days for BRWL, and ranges from 1 to 25 days for BRNL. Our results show that such differences are statistically significant in four out of 10 studied systems (ActiveMQ, Hadoop Common, HDFS, and Storm), where the effects range from small to large. Figure 5 further shows the beanplots that compare the density of the resolution time distribution between BRWL and BRNL. We limit the Y-axis to 15 days to better visualize the difference between the resolution time of BRWL and BRNL (most BRNL are resolved within 15 days). As illustrated in Figure 5, the distribution of the resolution time for BRNL generally has a long tail. In other words, most BRNL are resolved in a

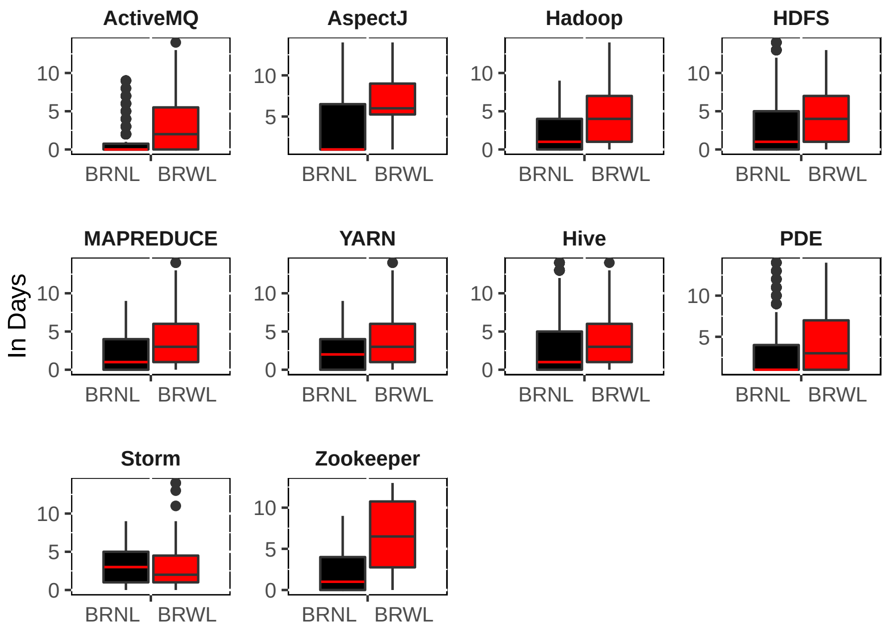


Fig. 6: Boxplots to illustrate the median resolution time (in days) for BRWL and BRNL in range of 15 days.

very short amount of time (within two to three days), and almost all BRNL are resolved within 15 days. BRWL, on the other hand, have more uniform distributions in the studied systems. To better illustrate this finding, Figure 6 shows the boxplots that compare the median resolution time between BRWL and BRNL in range of 15 days. BRNL are generally resolved in a shorter amount of time than that of BRWL.

Prior studies (Bettenburg *et al.*, 2008b; Zimmermann *et al.*, 2010) found that log snippets and stack traces are important debugging information in bug reports. However, even though such information is useful for debugging, we find that BRWL take more time to resolve compared to BRNL. Hence, we further investigate the possible factors that may increase the resolution time for BRWL. We first study where the logs are attached in bug reports. As discussed in Section 2, developers may request more logs in the Comments section of a bug report, which may take time for the reporter to provide and delay the bug fixing. Table 4 shows the percentage of BRWL with *logs only in the Description section* (i.e., BRWL-D) and BRWL with *logs in the Comments section* (i.e., BRWL-C, both BRWL with logs only in the Comments and BRWL with logs in both the Description and Comments), along with their respective median number of log lines and median resolution time. We find that the BRWL-C covers from 17% to 68% (an average of 43%) of BRWL. In addition, the median number of log lines in the Comments section is comparable to that of the Description section. For the median resolution time, however, BRWL-C require much more time to resolve (i.e., medians are 1.1 to 36.8 times slower) compared to that of BRWL-D. The Wilcoxon rank-sum test shows that the resolution time from BRWL-D is statistically significantly different from the BRWL-C ($p < 0.001$). We use Cliff's Delta to assess the magnitude of this difference, which results to a small effect size (i.e., $|d|$ is 0.31). We further examine the Spearman rank correlation between the number of log lines in the Comments section and the resolution time. Although the correlation is not strong, we find that there are some correlations between the bug resolution time and the number of log lines in the Comments section (0.20 across all studied systems). Our finding shows that it is common for developers to ask for more logs to diagnose a bug, and having more logs in the Comments section may increase bug resolution time. In other words, the initial-attached logs may be insufficient for debugging. Figure 4 illustrates an example of such cases. The bug report ZOOKEEPER-2982 highlights an Internet Protocol address (IP) resolution bug in the ZooKeeper server. Although the reporter initially added some stack traces in the bug description illustrating the root cause, he was later asked by the developer to provide the steps to reproduce the bug and some server logs to help the bug fix.

Different from other studied systems, our finding shows that, in Eclipse PDE and AspectJ, the bug reports with logs are resolved faster than the ones without. The median resolution time for BRWL and BRNL are 3 and 6 days for PDE, respectively, and the difference is statistically significant (p-value $< 0.05$) with a negligible effect size. The median resolution time for BRWL and BRNL are 14 and 25 days for AspectJ, respectively, and the difference is not statistically significant (p-value $= 0.89$). After some investigation, we find that, compared to other studied systems, Eclipse PDE and AspectJ have the least percentage of BRWL-C. As shown in Table 4, BRWL-C take more time to resolve. For PDE and AspectJ, there are only 23% and 33% of the bug reports that have logs in the Comments section, respectively.

Another factor that associates with the bug resolution time is the complexity of bug fixes. We further compare the complexity of the bug fixes between BRWL and BRNL. For each bug report, we compute the number of changed lines of code (i.e., the total number of additions and deletions). In general, we find that the median number of changed lines of code is 51 for BRWL and 30 for BRNL. We also calculate the non-parametric Wilcoxon rank-sum test to compare the number of changed lines between BRWL and BRNL. The Wilcoxon rank-sum test shows that BRWL is statistically significant different from BRNL in terms of changed lines ($p < 0.001$). To assess the magnitude of this difference, we use Cliff's Delta. The difference between the number of changed lines of code for BRWL and BRNL is negligible (i.e., $|d|$ is 0.12). In short, we find that the bug fixes for BRWL are larger than BRNL, which may be positively correlated with the longer fixing time of BRWL.

Table 4: A comparison of the number of log lines and median resolution time between BRWL that have logs only in Description (BRWL-D) and BRWL that have logs in Comments (i.e., BRWL-C, BRWL with logs only in Comments and BRWL with logs in both Description and Comments).

| **Project** | **# of BRWL-C** | **Median # of log lines** | **Median resolution time** | **# of BRWL-D** | **Median # of log lines** | **Median resolution time** |
|---|---|---|---|---|---|---|
| **ActiveMQ** | 40 (43%) | 28 | 221 | 52 (57%) | 21 | 6 |
| **AspectJ** | 15 (33%) | 6 | 84 | 30 (67%) | 6 | 11 |
| **Hadoop Common** | 72 (47%) | 12 | 8 | 80 (53%) | 13 | 7 |
| **HDFS** | 116 (57%) | 22 | 37 | 86 (43%) | 13 | 21 |
| **MapReduce** | 111 (58%) | 18 | 36 | 82 (42%) | 17 | 19 |
| **YARN** | 132 (49%) | 18 | 13 | 140 (51%) | 14 | 6 |
| **Hive** | 41 (32%) | 22 | 18 | 88 (68%) | 29 | 5 |
| **PDE** | 83 (23%) | 9 | 27 | 282 (77%) | 11 | 1 |
| **Storm** | 11 (17%) | 27 | 14 | 53 (83%) | 17 | 4 |
| **Zookeeper** | 32 (68%) | 16 | 116 | 15 (32%) | 25 | 46 |
| | **total:** 653 (42%) | **avg:** 18 | **avg:** 57 | **total:** 908 (58%) | **avg:** 17 | **avg:** 13 |

We find that BRWL takes more time to resolve (median ranges from 3 to 91 days) compared to BRNL (median ranges from 1 to 25 days). Our further investigation shows that the initially-attached logs may not be sufficient for debugging (i.e., developers often ask for more logs in the Comments section of a bug report), and the bug fixing size of BRWL is, in general, larger than BRNL (median is 51 vs 30 lines of code).

## 4.2 RQ2: Are There Overlaps Between Logged Classes and Fixed Classes?

**Motivation.** Logs illustrate important system run-time information. When debugging user-reported bugs, logs (i.e., either log snippets, stack traces, or both) are usually the only source of information that is available to developers (Fu *et al.*, 2014; Yuan *et al.*, 2010, 2012a). Developers need to manually analyze the logs to diagnose the problem. Hence, if the attached logs are unclear or insufficient, debugging can become even more time consuming and challenging (LaToza and Myers, 2010; Yuan *et al.*, 2010, 2012b). Even though prior studies have leveraged logs to assist bug localization (Moreno *et al.*, 2014; Wang and Lo, 2016; Wong *et al.*, 2014), it is still not clear about the direct effects of the logs and their possible limitations. In this RQ, we study the overlap between the logged classes (i.e., classes that generated the logs) and the fixed classes (i.e., classes where developers applied bug fixes). Our findings provide the empirical evidence on the importance and usefulness of providing additional tools and information to help developers in analyzing user-provided logs in bug reports.

**Approach.** Our goal is to study if there exist overlaps between the logged classes and the fixed classes (i.e., whether or not at least one of the fixed classes is the same as the classes that generated the user-reported logs). Our first step is to extract the logged classes from bug reports. As mentioned in Section 2, we capture the logs using regular expression. Specifically, we

Table 5: An overview of the bug reports with fixed classes overlapping with the logged classes. The average numbers are computed based on each bug report. The percentage of fixed classes located in logs is the ratio of the fixed classes in logs to the total fixed classes in bug report (# fixed classes in logs / # total fixed classes).

| **Project** | **# of BR with fixed classes located in logs** | **Avg. # of fixed classes per BR** | **Avg. # of logged classes per BR** | **% of fixed classes located in logs** |
|---|---|---|---|---|
| **ActiveMQ** | 25 (58%) | 2.3 | 15.3 | 41.6% |
| **AspectJ** | 23 (65%) | 2.0 | 5.5 | 33.7% |
| **Hadoop Common** | 87 (65%) | 2.4 | 6.8 | 50.0% |
| **HDFS** | 119 (71%) | 2.8 | 16.1 | 48.2% |
| **MapReduce** | 108 (70%) | 2.2 | 8.6 | 49.7% |
| **YARN** | 192 (79%) | 3.3 | 11.2 | 51.0% |
| **Hive** | 91 (75%) | 2.9 | 12.6 | 51.6% |
| **PDE** | 291 (81%) | 4.5 | 14.4 | 24.5% |
| **Storm** | 30 (55%) | 2.0 | 7.6 | 38.7% |
| **Zookeeper** | 29 (63%) | 2.2 | 6.7 | 46.2% |
| | **total:** 995 (73%) | **avg:** 2.7 | **avg:** 10.5 | **avg:** 43.5% |

look for log snippets by extracting log lines that contain timestamps (e.g., `17/07/14 13:31:58`), verbosity level (e.g., `INFO`), and fully-qualified class name (e.g., `org.apache.hadoop.mapred.TaskTracker`). We highlight stack traces in a similar fashion by using the `at` keyword, followed by a fully-qualified class name, method invocation, and line number. At the end of the first step, we get a list of fully-qualified class names covered in logs.

The next step is to extract the list of fixed classes for each bug report. We follow prior studies (Kim *et al.*, 2006; Śliwerski *et al.*, 2005) by linking the bug reports to the associated bug fixing commits using bug IDs. In the studied systems, developers are required to record the bug IDs in commit messages. Therefore, we use the `git log | grep BUG_ID[^\d]` command to find the corresponding bug fixing commits of a bug. Once we get these commits, we find the list of fixed Java files and compute for their fully-qualified class name from the `package` declaration statement (e.g., `package org.apache .hadoop.mapred.TaskTracker`). Finally, we compared the fixed classes that overlap with the logged classes. To note that both the logged classes and fixed classes are collected at outer class-level. To further refine our analysis, we exclude 191 bug reports that did not modify any existing Java classes. We then conduct a manual study on these bug fixes to examine the reason.

**Results.** ***Classes covered in user-reported logs provide a good indication of where the bug may be located.*** Table 5 shows the overview of the bug reports where the fixed classes have an overlap with the logged classes. We find that 88% (1,370/1,561) bug reports modified existing Java classes when fixing bugs. We further study the remaining 191 bug reports that did not modify any existing Java class later in this RQ. There are 73% (995/1,370) bug reports that have an overlap between the fixed classes and the logged classes. In other words, to a large extent, logs provide direct information for developers to diagnose and fix a bug. In addition, Table 5 shows the number of classes

Table 6: A comparison of the median resolution time for the bug reports with fixed classes located in logs and the ones without.

| **Project** | **# of BR with fixed classes located in logs** | **Median resolution time (days)** | **# of BR with no fixed classes located in logs** | **Median resolution time (days)** |
|---|---|---|---|---|
| **ActiveMQ** | 25 (58%) | **14** | 18 (42%) | **57** |
| **AspectJ** | 23 (65%) | 16 | 22 (35%) | 13 |
| **Hadoop Common** | 87 (65%) | 7 | 47 (35%) | 6 |
| **HDFS** | 119 (71%) | **18** | 48 (29%) | **26** |
| **MapReduce** | 108 (70%) | **11** | 46 (30%) | **26** |
| **YARN** | 192 (79%) | 10 | 52 (21%) | 7 |
| **Hive** | 91 (75%) | 7 | 30 (25%) | 6 |
| **PDE** | 291 (81%) | **3** | 70 (19%) | **6** |
| **Storm** | 30 (55%) | **4** | 25 (45%) | **25** |
| **Zookeeper** | 29 (63%) | **60** | 17 (37%) | **117** |
| | **total:** 995 (73%) | **avg:** 15 | **total:** 375 (27%) | **avg:** 29 |

covered in user-reported logs. We find that the user-reported logs often cover 5.5 to 16.1 unique classes and these logged classes have an overlap with 24.5% to 51.6% of the fixed classes. Given the fact that, on average, fixing a bug report requires only modifying 2 to 4.5 classes in the studied systems. Our finding shows that even without any advanced techniques, the user-reported logs may provide a good indication of the fixed classes. Furthermore, on some systems, the median resolution time is drastically reduced for bug reports that have class overlap. Table 6 shows the median resolution time for bug reports with class overlap and the ones without. For bug reports with class overlap, the resolution time can be reduced up to 6.3 times. However, as we also find, not all fixed classes are found in logged classes. Further improvement can be done to better assist developers. For instance, future research can develop tools to reconstruct the execution path based on the user-reported logs to assist developers with bug fixing as we observe cases where the fixed classes are located on the execution path.

Similar to the prior study conducted by Schroter *et al.* (2010), we further analyze the bug reports with fixed classes in stack traces (725/995) to study the position of the fixed class in the stack frames. Figure 7 shows an overview between the position of the fixed class in stack trace and the cumulative percentage of bug reports. We observe that 40% of the bug reports have the fixed class located at the first stack frame, 70% have the fixed class located within the top-5 stack frames, and more than 90% have the fixed class located within the top-15 stack frames. However, when we further analyze the relationship between the position of the fixed class and the resolution time of the bug report, the Spearman correlation is nearly zero (0.08). One potential reason is that bug reports are only marked as resolved or fixed after they have been tested, code-reviewed, and integrated into the production environment. There are many factors that can influence the resolution time (e.g., time of bug triage and replication). As the position of the fixed class is only relevant to the debugging process, its effect becomes less significant to the overall resolution

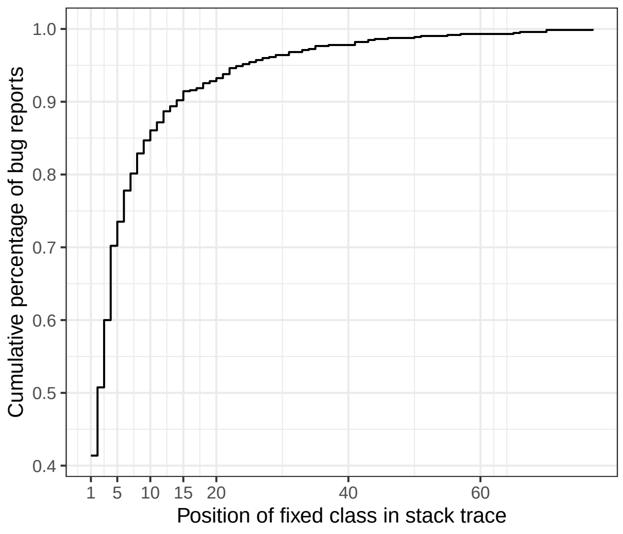


Fig. 7: Cumulative percentage of bug reports for the position of fixed class in stack trace.

time. Therefore, our finding shows that there is no clear correlation between the position of the fixed class in the stack frame and the bug resolution time.

Table 7 shows examples where there is an overlap between the logged classes and fixed classes. HADOOP-5233 (i.e., first row in Table 7) reports a bug where the reducer transits from `COMMIT_PENDING` to `RUNNING` state while it should wait for the commit response. The user-provided logs show the unexpected transition from `COMMIT_PENDING` state, generated by the *TaskTracer* class. The bug fix to HADOOP-5233 adds a conditional logic to ignore the progress update in the *TaskTracker* class whenever the state changes from `COMMIT_PENDING` to `RUNNING`. Thus, the logged class *Task-Tracer* overlaps with the fixed class. Other changes (i.e., the changes that occur in *JobInProgress*, *Task*, *TaskInProgress* and *TaskStatus*) make sure that the `COMMIT_PENDING` task entry is properly removed from the tracker. HDFS-10512 (i.e., second row in Table 7) describes a bug that triggers an unexpected `NullPointerException` in the *VolumeScanner* class (i.e., a volume scanner is responsible to scan block data to detect data corruptions) while reading for a volume variable through the *DataNode.reportBadBlocks* method call. The bug fix essentially added a conditional operator to verify whether the volume variable is `null` in the *DataNode* class. The changes to *FsDatasetImpl* and *VolumeScanner* are to adopt existing codes to the changes. In addition, a new test case is added to the *TestFsDatasetImpl* class to test the *DataNode.reportBadBlocks* method when the volume is null. The logged classes overlaps with the fixed classes *DataNode* and *VolumeScanner*.

Table 7: Examples of direct mapping between logged classes and fixed classes. Note that we simplify these examples by only showing the class name instead of the fully-qualitified class name.

| **Bug Report** | **Logs** | **Logged classes** | **Fixed classes** | **Overlaps** |
|---|---|---|---|---|
| HADOOP-5233 | ... 2009-02-12 08:35:36,417 INFO org.apache.hadoop.map -red.TaskTracker: Task attempt_200902120746_0297_r _000033_0 is in COMMIT_PENDING 2009-02-12 08:35:36,417 INFO org.apache.hadoop.map -red.TaskTracker: attempt_200902120746_0297_r_0000 33_0 0.33% reduce > sort | TaskTracker | JobInProgress Task TaskInProgress TaskStatus TaskTracker | TaskTracker |
| HDFS-10512 | ... 2016-04-07 20:30:53,831 ERROR org.apache.hadoop. hdfs.server.datanode.VolumeScanner: VolumeScanner (/dfs/dn, DS-89b72832-2a8c-48f3-8235-48e6c5eb5ab3) exiting because of exception java.lang.NullPointer -Exception at org.apache.hadoop.hdfs.server.datanode.DataNo -de.reportBadBlocks(DataNode.java:1018) at org.apache.hadoop.hdfs.server.datanode.Volume -Scanner$ScanResultHandler.handle(VolumeScanner .java:287) at org.apache.hadoop.hdfs.server.datanode.Volume -Scanner.scanBlock(VolumeScanner.java:443) at org.apache.hadoop.hdfs.server.datanode.Volume -Scanner.runLoop(VolumeScanner.java:547) at org.apache.hadoop.hdfs.server.datanode.Volume -Scanner.run(VolumeScanner.java:621) ... | DataNode VolumeScanner | DataNode FsDatasetImpl TestFsDatasetImpl VolumeScanner | Datanode VolumeScanner |

We further manually examine the bug reports in which no existing Java classes were modified in the bug fix. We manually study a statistically representative random sample of 162 bug reports out of the 191 bug reports (with a confidence level of 95% and a confidence interval of 3%). We classify these bug reports into four categories: *non-Java code changes*, *configuration file changes*, *only added new Java classes*, and *incorrect commit*. *Non-Java code changes* (85/162) are bug fixes performed on programming source code files other than `.java`. Such source code files are usually system-specific. For example, in HIVE, a big majority of these bug reports changed test query files (*.q*) and test query result files (*.q.out*). *Configuration file changes* (65/162) are bug fixes that only modified configuration files, such as managing dependencies in *.xml* file for Maven projects. *Only added new Java classes* (8/162) are bug reports where only new Java classes were added to the studied system, and no existing Java class was modified. For such bugs, it is impossible for the logs to be mapped to a new fixed class that is yet to exist in the system. We also find that this type of bug fixes is uncommon and developers often modify other configuration files to adopt the newly added Java classes. Finally, *incorrect commit* (4/162) consists of bug reports where bug fixes were committed with the incorrect bug ID. In short, our findings show that it is common for developers to modify files that are written in different programming languages, and some bugs can actually be fixed by modifying configuration files. Future studies should consider the polyglot nature of modern software systems and the importance of configuration files in fixing bugs.

The fixes to 88% (1,370/1,561) of the BRWL included modifications to existing Java classes. We find that 73% (995/1,370) of the bug reports have overlaps between the logged classes and fixed classes. Depending on the quality of the logs, the logged classes can locate up to 51.6% (44% on average) of the fixed classes. Although the user-provided logs provide a good indication on the bug fixing locations in some situations, there is still an average of 56% of the fixed classes that have no overlap with the logged classes.

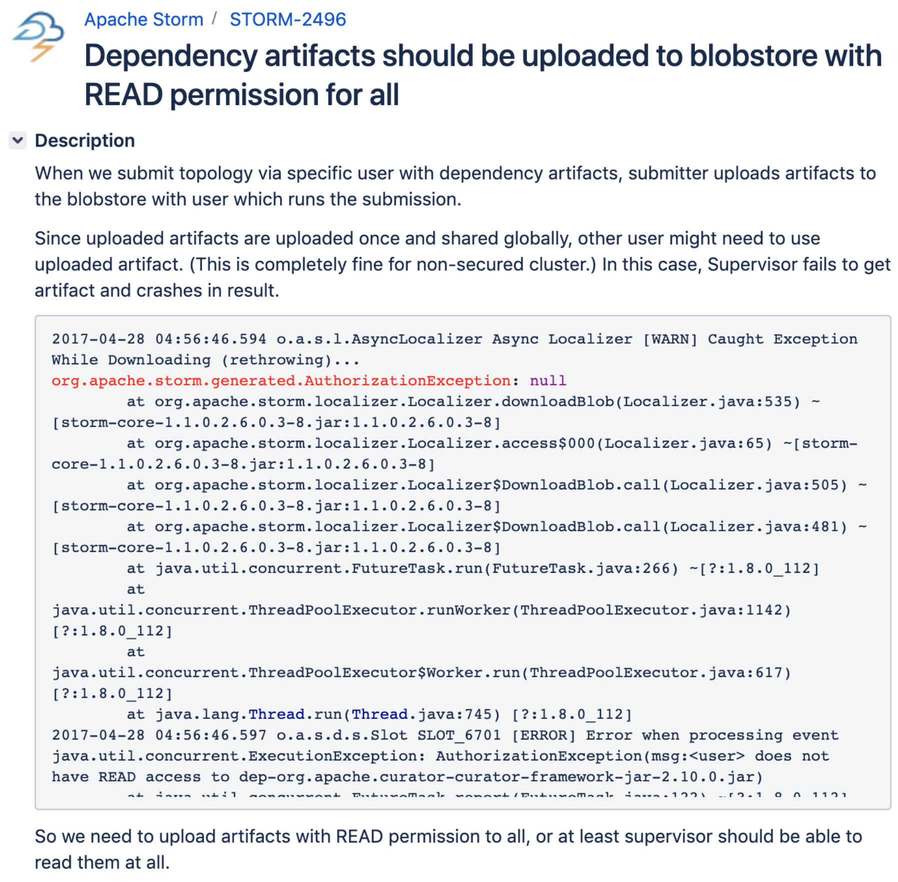
## 4.3 RQ3: Why do some fixed classes have no overlap with the logged classes?

**Motivation.** Unlike bugs that are uncovered during development phases, many user-reported bugs are difficult to reproduce and often lack test cases (Tucek *et al.*, 2007; Yuan *et al.*, 2012a, 2014). In such cases, developers rely on logs during the debugging process (Yuan *et al.*, 2010, 2011, 2012a). However, as we found in RQ2, even though there is an overlap between logged classes and fixed classes, there are some bugs where the user-provided logs cannot help identify fixed classes (27%, 375/1,370) after excluding the bug reports that had no modified Java class in bug fixes. Therefore, in this RQ, we manually investigate the reasons why certain user-provided logs fail to find the fixed classes (i.e., cannot help identify any fixed classes). Our findings may provide insights on helping researchers and practitioners improve the current logging practice.

**Approach.** We manually study the bug reports in which the logs could not help identify fixed classes at all. From RQ2, we find that 27% (375/1,370) of bug reports have no overlaps between the logged classes and fixed classes. Hence, we then manually study 278 out of 375 such bug reports to achieve a confidence level of 95% and a confidence interval of 3% (Moore *et al.*, 2009). The first author of the paper manually studied the bug reports. The first author examined the bug reports, the attached logs, the bug fixes, source code classes, and the development history (e.g., prior commits) to understand the reason. The first author took notes while studying each bug report. At the end of the process, we uncovered a list of categories for which there was no direct mapping between logged classes and fixed classes. We then revisited and assigned each bug report to the uncovered categories. The second author of the paper helped verify the assigned categories and any discrepancy (e.g., on which category the bug report belongs to) is discussed until there is a consensus.

**Results.** In total, we uncovered two categories of reasons for which there was no direct mapping between the logged classes and fixed classes. Below, we discuss each category in detail.

***Logs that show the failure but not the fault (i.e., the root cause) (266/278).*** We find that reporters in most of the 278 studied bug reports attached related logs to the bug, but the logged classes do not have an overlap



Apache Storm / STORM-2496

Dependency artifacts should be uploaded to blobstore with READ permission for all

Description

When we submit topology via specific user with dependency artifacts, submitter uploads artifacts to the blobstore with user which runs the submission.

Since uploaded artifacts are uploaded once and shared globally, other user might need to use uploaded artifact. (This is completely fine for non-secured cluster.) In this case, Supervisor fails to get artifact and crashes in result.

```
2017-04-28 04:56:46.594 o.a.s.l.AsyncLocalizer Async Localizer [WARN] Caught Exception
While Downloading (rethrowing)...
org.apache.storm.generated.AuthorizationException: null
        at org.apache.storm.localizer.Localizer.downloadBlob(Localizer.java:535) ~
[storm-core-1.1.0.2.6.0.3-8.jar:1.1.0.2.6.0.3-8]
        at org.apache.storm.localizer.Localizer.access$000(Localizer.java:65) ~[storm-
core-1.1.0.2.6.0.3-8.jar:1.1.0.2.6.0.3-8]
        at org.apache.storm.localizer.Localizer$DownloadBlob.call(Localizer.java:505) ~
[storm-core-1.1.0.2.6.0.3-8.jar:1.1.0.2.6.0.3-8]
        at org.apache.storm.localizer.Localizer$DownloadBlob.call(Localizer.java:481) ~
[storm-core-1.1.0.2.6.0.3-8.jar:1.1.0.2.6.0.3-8]
        at java.util.concurrent.FutureTask.run(FutureTask.java:266) ~[?:1.8.0_112]
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
[?:1.8.0_112]
        at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
[?:1.8.0_112]
        at java.lang.Thread.run(Thread.java:745) [?:1.8.0_112]
2017-04-28 04:56:46.597 o.a.s.d.s.Slot SLOT_6701 [ERROR] Error when processing event
java.util.concurrent.ExecutionException: AuthorizationException(msg:<user> does not
have READ access to dep-org.apache.curator-curator-framework-jar-2.10.0.jar)
```

So we need to upload artifacts with READ permission to all, or at least supervisor should be able to read them at all.

Fig. 8: An example bug report (**STORM-2496**) that shows the reporter attached logs to illustrate unexpected behaviors (i.e., failure). The bug fix was applied in a related class (i.e., *DependencyUploader*), but the class is not shown in the stack trace.

Table 8: Percentage of bug reports and fixed classes located at different distances, where the distance is calculated as the shortest path between the logs and fixed classes in terms of class invocations. When there is no path between the logs and fixed classes, the fixed classes are marked as *unreachable*.

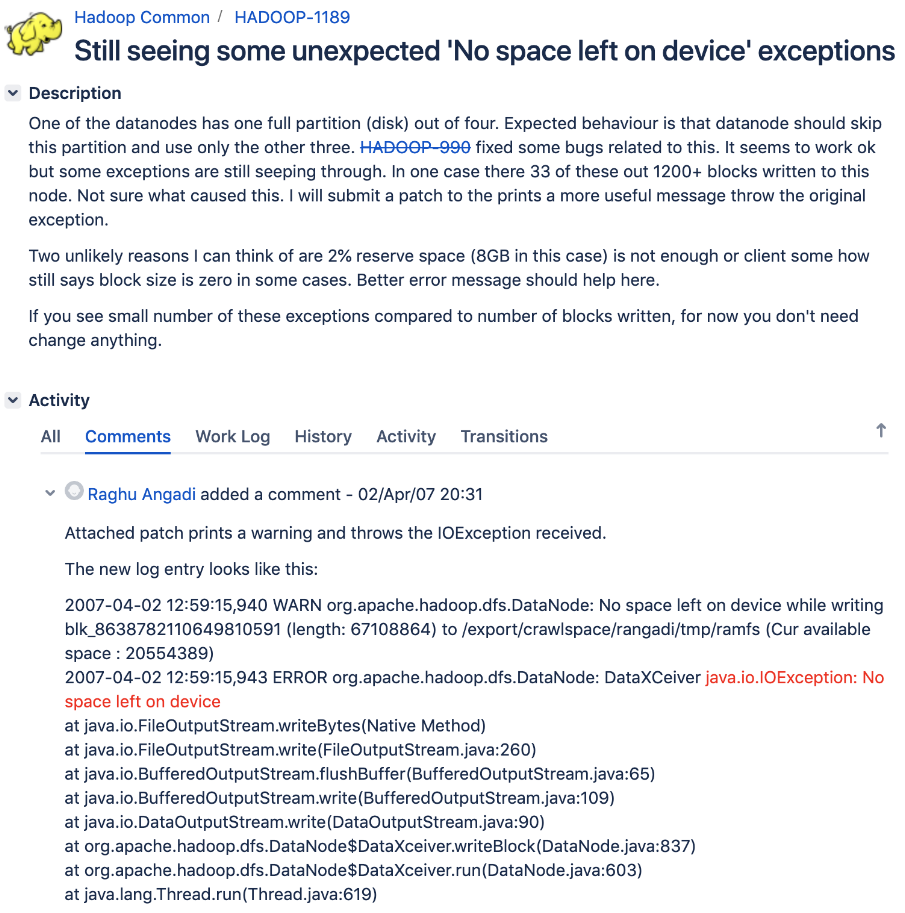
| # BR | | | | # fixed classes | | | |
|---|---|---|---|---|---|---|---|
| **total** | **dist=1** | **dist>1** | **unreachable** | **total** | **dist=1** | **dist>1** | **unreachable** |
| 266 | 61 (23%) | 13 (5%) | 192 (72%) | 564 | 83 (15%) | 19 (3%) | 462 (82%) |

with the fixed classes. In all the cases that we manually studied, the logs are reported to illustrate an unexpected behavior (i.e., the failure (IEEE, 2020)). The majority of the cases (i.e., 202 bug reports) are related to stack traces. As stack traces are used to provide debugging information at the point of failure, the faulty classes (i.e., the cause of the bug) do not fall into the stack frames of the stack traces. Figure 8 shows an example. In STORM-2496, a reporter attached stack traces to show the failure *AuthorizationException* when users upload dependency artifacts. In this stack trace, we see the list of stack frames leading to the exception and the state of the user's access permission being null. However, *DependencyUploader*, the essential class that manages permissions and where the bug fix was applied, is not shown in the logs. The reporter also did not attach the logs that happened before the exception, which may show the execution path that led to the exception and help locate the root cause.

Similar to the prior study by Moreno *et al.* (2014), we further analyze the shortest path in the call graph between the fixed classes and classes found in the logs at class level. Although some user-provided logs cannot help to identify any fixed classes, we want to investigate how far away the logged classes are from the fixed classes in the system. Thus, we further analyze the distance between the fixed classes and the logged classes. First, we select the commit prior to the bug fixing commit as our affected version (i.e., the bug is still unresolved). Then, we derive the system call graph on the affected version using JavaParser [1]. JavaParser is a static analysis tool that transforms the source code to Abstract Syntax Tree (AST) for Java applications. We traverse the method calls in the ASTs to uncover all the paths in the call graph. Once the paths are generated, we calculate the distance for the shortest path, if it exists, between the fixed classes and the logged classes by applying depth-first search.

For the 266 bug reports that belong to this category, 61 (23%) bug reports have fixed classes that are one distance away from the classes shown in the logs, 13 (5%) are two distance or further, and 192 (72%) bug reports have fixed classes that are unreachable from the classes in the logs. The result implies that 28% of the studied bug reports have the fixed classes that are reachable (i.e., one distance away or further in the call graph) to the classes in logs. Besides, in terms of the number of fixed classes in stack traces, our finding shows that up to 18% of the fixed classes (15% that are one distance away from the logged classes, and 3% are two distance or further) can be located in the call graph. The result shows that even for some of the bug reports which have no overlap between the logged classes and fixed classes, the execution path re-constructed from the logged classes may be used to suggest the potential bug fixing locations.

There are a few other reasons where the fixed classes cannot be located using the attached logs. Figure 9 shows an example of such bug reports. In this example, *DataNode* throws *IOException* when one of the partitions does not have enough remaining disk space. The logs show the execution of blocks (i.e., in a distributed storage system, blocks are essentially chunks of files that are stored across DataNodes) when writing to *DataNode*. The bug occurs inside the



Still seeing some unexpected 'No space left on device' exceptions

One of the datanodes has one full partition (disk) out of four. Expected behaviour is that datanode should skip this partition and use only the other three. ~~HADOOP-990~~ fixed some bugs related to this. It seems to work ok but some exceptions are still seeping through. In one case there 33 of these out 1200+ blocks written to this node. Not sure what caused this. I will submit a patch to the prints a more useful message throw the original exception.

Two unlikely reasons I can think of are 2% reserve space (8GB in this case) is not enough or client some how still says block size is zero in some cases. Better error message should help here.

If you see small number of these exceptions compared to number of blocks written, for now you don't need change anything.

Raghu Angadi added a comment - 02/Apr/07 20:31

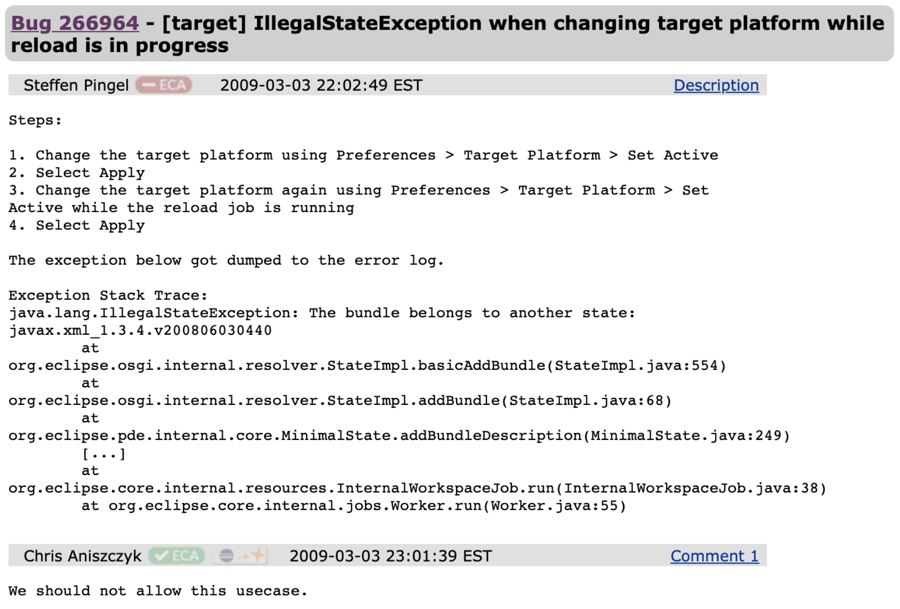
Attached patch prints a warning and throws the IOException received.

The new log entry looks like this:

```
2007-04-02 12:59:15,940 WARN org.apache.hadoop.dfs.DataNode: No space left on device while writing
blk_8638782110649810591 (length: 67108864) to /export/crawlspace/rangadi/tmp/ramfs (Cur available
space : 20554389)
2007-04-02 12:59:15,943 ERROR org.apache.hadoop.dfs.DataNode: DataXCeiver java.io.IOException: No
space left on device
at java.io.FileOutputStream.writeBytes(Native Method)
at java.io.FileOutputStream.write(FileOutputStream.java:260)
at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)
at java.io.BufferedOutputStream.write(BufferedOutputStream.java:109)
at java.io.DataOutputStream.write(DataOutputStream.java:90)
at org.apache.hadoop.dfs.DataNode$DataXceiver.writeBlock(DataNode.java:837)
at org.apache.hadoop.dfs.DataNode$DataXceiver.run(DataNode.java:603)
at java.lang.Thread.run(Thread.java:619)
```

Fig. 9: An example bug report (**HADOOP-1189**) that highlights the insufficient disk space left in one of the partitions. The bug fix updated the *FSDataset* class which is not shown in the logs (but based on our manual study, the *FSDataset* class is invoked between the first log and the second log).

*getAvailable* method, from the *FSDataset* class, that incorrectly calculates the available space. The *getAvailable* method was executed as part of the execution between the first log snippet carrying the message of "*No space left on device while writing ...*" and the stack trace throwing the *IOException*. However, the class (i.e., *FSDataset*) is not recorded in the stack trace since calls to the class have returned before throwing exceptions, so are no longer available in the stack. Since logs are expensive to maintain and monitor (Li *et al.*, 2019; Yuan *et al.*, 2011), developers may need to prioritize on logging the essential code snippets. Hence, some code snippets may be ignored and not logged. As shown in the previous example, an important code snippet was hidden between two logging statements. One potential direction for future research is to focus on



**Bug 266964 - [target] IllegalStateException when changing target platform while reload is in progress**

Steffen Pingel ECA 2009-03-03 22:02:49 EST Description

```
Steps:

1. Change the target platform using Preferences > Target Platform > Set Active
2. Select Apply
3. Change the target platform again using Preferences > Target Platform > Set
Active while the reload job is running
4. Select Apply

The exception below got dumped to the error log.

Exception Stack Trace:
java.lang.IllegalStateException: The bundle belongs to another state:
javax.xml_1.3.4.v200806030440
        at
org.eclipse.osgi.internal.resolver.StateImpl.basicAddBundle(StateImpl.java:554)
        at
org.eclipse.osgi.internal.resolver.StateImpl.addBundle(StateImpl.java:68)
        at
org.eclipse.pde.internal.core.MinimalState.addBundleDescription(MinimalState.java:249)
        [...]
        at
org.eclipse.core.internal.resources.InternalWorkspaceJob.run(InternalWorkspaceJob.java:38)
        at org.eclipse.core.internal.jobs.Worker.run(Worker.java:55)
```

Chris Aniszczyk ECA 2009-03-03 23:01:39 EST Comment 1

```
We should not allow this usecase.
```

Fig. 10: An example bug report (**Eclipse PDE Bug 266964**) where the fixed classes are unreachable through the call graph.

reconstructing the execution path among logs and uncover the hidden paths between logs to further assist debugging.

Our finding indicates that reporters often only attach debugging information for the point of failure (e.g., stack traces). Although such information is helpful, there is a missing link between the failure and the root cause of the problem in the source code. Reporters may consider attaching additional logs (e.g., log snippets) that show the execution of the system in addition to stack traces. Additional research is required to help reporters provide missing logs in bug reports that complete the execution information and help developers with debugging the problem.

To better illustrate the cases where the fixed classes are unreachable through the call graph, the figure 10 shows such example. The bug report Eclipse PDE 266964 shows an *IllegalStateException* when modifying the preferred platform. This error is due to the user job that keeps running while the user switches the target platform. The stack trace shows that the *Worker* class continues to process the user job which leads to the *IllegalStateException*. The developers discussed in the comments that such use cases should not be allowed. The fixed classes were TargetPlatformPreferencePage2, TargetEditor and LoadTargetDefinitionJob. The fix ensured that any existing jobs are cancelled before the target platform switches. In such cases, the bug fix occurs in a small workflow change of the system, and it is almost impossible for developers to show such details in logs.

***Code evolution (12/278).*** We find that sometimes the source code that generates the logs no longer exists. In other words, the logs that the reporters provide are from an older version of the system. The logging statements or the source code class may have been removed during evolution. In such cases, developers may have additional challenges in understanding and fixing the bug. In addition, we find that 28.1% (323/1,151) of the studied bug reports do not have values for the *Affects Version* field (i.e., entered by the reporter or developers to indicate which versions they observed the bug). Even if the bug reports have *Affects Version*, only 32.4% (268/828) of the bug reports have the same *Fix Version* as the *Affects Version*. Note that we exclude PDE and AspectJ bug reports from this analysis since the *Fix Version* field is not available on Bugzilla. Namely, developers often debug and perform the fix on a different version of the code and not on the reported *Affects Version*. Our finding highlights that version information is essential for a high-quality bug report. Therefore, reporters are strongly suggested to include version information of the buggy system when submitting a bug report. Future studies should also be conducted to help developers analyze such bug reports by taking the past development history (e.g., prior source code changes) into consideration, since the fixes may need to be applied to newer versions of the system.

Our manual study finds that some user-provided logs only show the unexpected behavior (i.e., failure), but do not show the root cause of a bug nor the execution that led to the failure. Reporters should consider attaching additional logs to assist in debugging. In addition, some attached logs are from prior versions of the systems and can no longer be found in the source code. Future research is required to utilize prior source code changes as an important debugging hint for developers when analyzing bug reports.

## 5 Discussion and Implication of Our Findings

In this section, we summarize our findings and provide some discussion and implications.

**More research and supports are needed for logging code evolution.** In our manual study in RQ3, we find that some user-provided logs (i.e., either stack traces, log snippets, or both) can no longer be found in the version that developers are working on. Different from a prior study (Yuan *et al.*, 2012b), we found that it is not uncommon for logging statements or methods in stack traces to be removed from the source code. If developers are not familiar with the system, such logging statement changes can cause additional challenges during debugging. Future studies should consider analyzing software development history and help developers locate the user-provided logs, for which the corresponding logging statements/methods were deleted or moved. In addition, for reporters, it is essential to provide the version information of the system when reporting a bug.

**Reporters need additional assistance on providing logs in bug reports.** Although logs provide important debugging information for developers, reporters may not be able to provide accurate logs that can illustrate the problem. For example, we find that reporters may attach incomplete logs or logs that only illustrate the exception. Hence, future studies should also consider helping reporters provide more accurate logs that can better assist debugging. One potential direction is to study the part of system execution that is not illustrated in the reported logs to find the missing link between the failure and the root cause of the problem.

**Future studies could consider using execution paths that are reconstructed from readily-available runtime data to provide additional debugging supports.** We find that, even though the quality of user-provided logs may not be perfect, these logs still provide a good indication of the fixed classes. Our finding highlights a potential direction that may further assist developers with debugging. For example, future studies may leverage logs to re-construct the execution paths between each log message or stack frame. For instance, as shown in Figure 9, although the fixed class is invoked on the execution path leading to the bug, but it does not directly appear in the reported stack trace. Therefore, to further assist developers in debugging, additional research is needed to leverage user-provided logs in re-constructing the execution paths leading to failures.

## 6 Threats to Validity

In this section, we discuss the threats to validity related to this study.

### 6.1 External Validity.

Threats to external validity are related to the generalizability of our findings. To increase the generalizability of our study, we conduct our case study on 10 large-scale open source systems that vary in size and infrastructures (e.g., data warehouse, real-time computation system, distributed file system). These systems are actively maintained and widely used. Although all the systems are Java-based, our approach is not limited to Java systems. We present our approach in a generic way that can easily be adapted to fit systems in other programming languages (e.g., by changing the regular expression). To reduce the external threat to validity, we include systems from different domains, ranging from databases to software development tools. We found that the results are similar across the studied systems. However, other system types, such as mobile applications, may use logs differently (e.g., for in-house debugging (Zeng *et al.*, 2019)) and our findings may not hold. Future studies are encouraged to conduct the analysis on systems in more diverse domains to improve the generalizability of our findings. For RQ3, we mitigate the sampling bias by ensuring the sample falls into a confidence level of 95% with a confidence interval of +/- 3%. When sampling for our manual data set, we carefully respect the sample size of each studied system and sampled proportionally according to the number of bug reports per system.

## 6.2 Internal Validity.

Threats to internal validity are related to experimenter errors and bias. Our study shows that the results of direct mapping between logged classes and fixed classes highly depend on the quality of user-provided logs. Thus, the extracted logs are an internal threat to the validity of our study. To mitigate this threat, we choose 10 systems that vary in software maturity, to better observe the difference in log quality of each studied system.

Another threat to internal validity is that we use bug IDs in commit messages to identify bug fixing commits. Although the developers in the studied systems are required to provide bug IDs in commit messages as part of the development guideline, there may still be some mistakes. For example, in our manual study in RQ3, we found a few cases where developers made a typo when providing bug IDs in the message. Nevertheless, we find such cases to be rare, and based on our manual study on a statistically representative sample, the heuristic has a very high precision (99%).

Another threat to internal validity is the way in which we collected the bug reports with logs. Typically, reporters attach logs in the bug description or as comments. Sometimes, when the logs are too long, reporters may upload them as attachments. Therefore, bug reports with logs might also include those that have log files in attachments. We further investigate this possibility, and find only a small number of the reporters upload logs as attachment (i.e., in 51 out of 8,849 bug reports, log files were added as attachment), which limits the impact of this threat.

In our study, we selected bug reports with priority *Major* or higher because bug reports with a lower priority may have less of an impact on the overall quality of the system. Moreover, these bug reports are less likely to be fixed. For example, we find that only 14% of the bug reports with logs marked as “Minor” or less were fixed in Hive, 13% in Hadoop Common and Storm, and 12% in MapReduce. Therefore, we follow prior studies (Chen and Jiang, 2017; Chen *et al.*, 2014; Yuan *et al.*, 2014) and focus our analysis on the bug reports with priority *Major* or higher.

## 6.3 Construct Validity.

In this paper, we have two manual studies. One investigates the reasons why some bug reports had no modification on existing Java files. The other one studies the reasons why some bug reports have no overlaps between the logged classes and fixed classes. Human biases may be introduced. To reduce the bias of our analysis, we have a second author to verify the assigned categories and any discrepancies are discussed until consensus is reached.

## 7 Related Work

In this section, we discuss related work in three areas: analyzing bug reports for debugging, debugging and maintaining software systems, and log analysis.

**Analyzing Bug Reports for Debugging.** Prior studies found that bug reports are essential for debugging (Anvik *et al.*, 2006; Bettenburg *et al.*, 2008a,b). In particular, Bettenburg *et al.* (Bettenburg *et al.*, 2008b) found that stack traces and steps to reproduce bugs are important for a good quality bug report. Similar to their findings, we found that logs provide a good indication of where a bug may be located. However, more often, we found that there is some missing information in the logs that may prevent developers from using the logs to locate bugs. Due to the rich information in bug reports, some studies proposed approaches to locate bugs in the source code by using text information in bug reports (Bhagwan *et al.*, 2018; Chaparro *et al.*, 2017; Dao *et al.*, 2017; Lam *et al.*, 2017; Liu *et al.*, 2016; Loyola *et al.*, 2018; Rahman and Roy, 2018; Saha *et al.*, 2013; Sisman and Kak, 2012; Wang and Lo, 2016; Zhou *et al.*, 2012). Wang and Lo (2016) and Saha *et al.* (2013) also found that different parts of bug reports (e.g., title and description) may provide more information to help locate the bugs in the source code. Different from prior studies that focus on developing approaches to help locate bugs by leveraging bug reports, we performed an empirical study to provide insights to improve bug localization, e.g., leveraging execution paths that are re-constructed from user-provided logs to provide additional information for identifying bugs.

**Debugging and Maintaining Software Systems By Leveraging Logs.** Logs, including both system execution logs and stack traces, are commonly used for understanding system execution (Chen *et al.*, 2016; Zhao *et al.*, 2014), maintaining software (Chen and Jiang, 2017; Yuan *et al.*, 2012b), testing (Chen *et al.*, 2018, 2017; Li *et al.*, 2018), and debugging (Yuan *et al.*, 2010, 2011). Prior studies (Chen and Jiang, 2017; Yuan *et al.*, 2012b) found that developers continuously improve logging code in software systems to assist in diagnosing production bugs. Li *et al.* (2020a) found that developers consider various benefits and costs when adding logging statements. These log messages are often the only information that is available for diagnosing production bugs (Yuan *et al.*, 2010, 2012a). Yuan *et al.* (2011, 2012a) tried to improve log messages (e.g., record values for important variables) to assist developers in diagnosing production bugs. Yuan *et al.* (2010) proposed a technique to assist developers with debugging by leveraging system runtime logs. However, the authors themselves manually evaluated their technique on only eight production bugs. Other studies apply machine learning techniques to identify anomalies in the log messages, which may be an indication of possible problems (Chen *et al.*, 2017; Lin *et al.*, 2016; Xu *et al.*, 2009). Hassani *et al.* (2018) analyzes bug reports that are related to logs (e.g., log levels or log messages) and provided a tool to automatically detect log-related issues. Schroter *et al.* (2010) compared the resolution time between the bug reports where the fixes are on the stack traces and the bug reports where the fixes are not on the stack trace. They found that the bug reports where the fixes are on the stack trace are fixed faster. Li *et al.* (2019) proposed an automated static analysis tool to identify duplicated logging statements code smells. Different from prior studies, we focus on studying the user-provided logs in bug reports, where the quality of the logs depends on the reporter (Li *et al.*, 2020b). We found that although the user-provided logs can help to debug by highlighting the logged classes to some extent, there are still some challenges. We manually studied and documented the challenges that we found in user-provided logs and provide future research directions.

The prior study by Moreno *et al.* (2014) found that 94.8% of the bug report can be fixed through the logged classes or the classes that are reachable in code structure to the logged classes. More specially, 64.5% (100/155) of the bug reports have the fixed classes at distance zero (i.e., where logged classes are overlapping with the fixed classes), 30.3% (47/155) at distance one or further, and 5% (8/155) are unreachable. This implies that, based on the 55 bug reports with logged classes that are not directly mappable, 15% (8/55) are unreachable with the other 85% (47/55) are reachable at distance one or further. Our result is different from that of the prior study. There are a few potential factors that lead to this difference in the results. Our bug reports are collected from 10 open-systems systems, where 9 out of 10 are different from the prior study. In our research, we focus on studying the bug reports with logged classes that are not directly mappable, thus we collected a larger sample size (i.e., 266 bug reports vs 55 bug reports). In addition, we include both log snippets and stack traces in our study and the prior study only considers stack traces. Nevertheless, our finding is similar to that of the prior study, which confirms the usefulness of the classes that are reachable (i.e., in call graph) from the logged classes in bug localization.

## 8 Conclusion

Logs in bug reports provide important information for developers to diagnose and fix the reported problems. However, due to privacy or technical constraints, users often do not provide the entire logs in a bug report. Therefore, the user-reported logs may be incomplete or inaccurate. In this paper, we conduct an empirical study on the user-provided logs in bug reports. In particular, we study the usefulness of the logs and potential challenges that developers may encounter when analyzing such logs. We conduct our case study on 10 large-scale open-source systems: ActiveMQ, AspectJ, Hadoop Common, HDFS, MapReduce, YARN, Hive, PDE, Storm, and Zookeeper. We find that: 1) bug reports with logs (BRWL) often take a longer time to resolve compared to bug reports without logs (BRNL). Our further analysis finds that developers often require additional logs in the Comments section of a bug report, which delays the bug fixes. In addition, the fixes of BRWL are more complex (i.e., modify more lines) than that of BRNL. 2) Most bug reports (73%) have an overlap between the logged classes that generate the reported logs in bug reports and their corresponding fixed classes, and the logged classes cover 38.7% to 51.6% of the fixed classes across our studied systems. Our results show that even without any advanced techniques, the user-reported logs may provide a good indication of the fixed classes. However, there is still a large number of bug reports where there is no overlap between the logged and fixed classes. 3) Our manual study finds that many logs only show the point of failure (e.g., exception) and not the actual root cause. In addition, some logging statements are removed in the source code as the system evolves, which may cause challenges in analyzing the logs. In summary, our empirical findings illustrate the usefulness of logs in bug reports and unveil the potential challenges. We also highlight future research directions on helping practitioners with attaching logs in bug reports and approaches to better analyze logs (e.g., consider using execution paths that are re-constructed from user-provided logs to provide additional debugging supports).

## References

Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 361–370.

Apache (2019). Aapache JIRA. Last accessed: Feb. 1, 2019.

Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. (2008a). Duplicate bug reports considered harmful... really? In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, ICSM '18.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2008b). What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*.

Bhagwan, R., Kumar, R., Maddila, C. S., and Philip, A. A. (2018). Orca: Differential bug localization in large-scale services. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 493–509. USENIX Association.

Bianchi, F. A., Pezzè, M., and Terragni, V. (2017). Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 705–716.

Cao, Y., Zhang, H., and Ding, S. (2014). Symcrash: Selective recording for reproducing crashes. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 791–802.

Chaparro, O., Florez, J. M., and Marcus, A. (2017). Using observed behavior to reformulate queries during text retrieval-based bug localization. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*, ICSME '17, pages 376–387.

Chen, B. and Jiang, Z. M. (2017). Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation. *Empirical Software Engineering*, **22**(1), 330–374.

Chen, B., Song, J., Xu, P., Hu, X., and Jiang, Z. M. J. (2018). An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, pages 305–316.

Chen, T.-H., Nagappan, M., Shihab, E., and Hassan, A. E. (2014). An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 82–91.

Chen, T.-H., Shang, W., Hassan, A. E., Nasser, M., and Flora, P. (2016). Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 666–677.

Chen, T.-H., Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2017). Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 243–252.

Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, **114**(3), 494–509.

Dao, T., Zhang, L., and Meng, N. (2017). How does execution information help with information-retrieval based bug localization? In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 241–250.

Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D., and Xie, T. (2014). Where do developers log? an empirical study on logging practices in industry. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE-SEIP '14, pages 24–33.

Hassani, M., Shang, W., Shihab, E., and Tsantalis, N. (2018). Studying and detecting log-related issues. *Empirical Software Engineering*.

IEEE (2020). Ieee definitions. `https://standards.ieee.org/standard/610_12-1990.html`. Last accessed March 23 2020.

Jin, W. and Orso, A. (2012). Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484.

Kim, S., Zimmermann, T., Pan, K., and Whitehead, E. J. J. (2006). Automatic identification of bug-introducing changes. In *Proc. of the 21st Int. Conference on Automated Software Engineering (ASE*.

Lam, A. N., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. (2017). Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 218–229.

LaToza, T. D. and Myers, B. A. (2010). Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on*

*Software Engineering*, ICSE '10, pages 185–194.

Li, H., Chen, T.-H. P., Hassan, A. E., Nasser, M., and Flora, P. (2018). Adopting autonomic computing capabilities in existing large-scale systems: An industrial experience report. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 1–10.

Li, H., Shang, W., Adams, B., Sayagh, M., and Hassan, A. E. (2020a). A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering*.

Li, Z., Chen, T.-H. P., Yang, J., and Shang, W. (2019). DLfinder: Characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 152–163.

Li, Z., Chen, T.-H., and Shang, W. (2020b). Where shall we log? studying and suggesting logging locations in code blocks. In *Proc. of the 35rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y., and Chen, X. (2016). Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 102–111.

Liu, B., Lucia, Nejati, S., Briand, L. C., and Bruckmann, T. (2016). Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability*, **26**(6), 431–459.

Loyola, P., Gajananan, K., and Satoh, F. (2018). Bug localization by learning to rank and represent bug inducing changes. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, pages 657–665.

Moore, D., MacCabe, G., and Craig, B. (2009). *Introduction to the Practice of Statistics*. W.H. Freeman and Company.

Moreno, L., Treadway, J. J., Marcus, A., and Shen, W. (2014). On the use of stack traces to improve text retrieval-based bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 151–160. IEEE.

Rahman, M. M. and Roy, C. K. (2018). Improving bug localization with report quality dynamics and query reformulation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 348–349.

Romano, J., Kromrey, J. D., Coraggio, J., and Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33.

Saha, R. K., Lease, M., Khurshid, S., and Perry, D. E. (2013). Improving bug localization using structured information retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 345–355.

Satvat, K. and Saxena, N. (2018). Crashing privacy: An autopsy of a web browser's leaked crash reports. *CoRR*, **abs/1808.01718**.

Schroter, A., Schröter, A., Bettenburg, N., and Premraj, R. (2010). Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121. IEEE.

Shang, W., Jiang, Z. M., Hemmati, H., Adams, B., Hassan, A. E., and Martin, P. (2013). Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 402–411.

Sisman, B. and Kak, A. C. (2012). Incorporating version histories in information retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 50–59.

Soltani, M., Panichella, A., and Van Deursen, A. (2018). Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, pages 1–1.

Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. (2007). Triage: Diagnosing production run failures at the user's site. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 131–144.

Wang, S. and Lo, D. (2016). Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, **28**(10), 921–942.

Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., and Mei, H. (2014). Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, page 181–190.

Wu, R., Zhang, H., Cheung, S.-C., and Kim, S. (2014). Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 204–214.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, Big Sky, Montana, USA. ACM.

Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S. (2010). Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 143–154.

Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. (2011). Improving software diagnosability via log enhancement. In *ASPLOS '11: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 3–14, Newport Beach, California, USA. ACM.

Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., and Savage, S. (2012a). Be conservative: Enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 293–306.

Yuan, D., Park, S., and Zhou, Y. (2012b). Characterizing logging practices in open-source software. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 102–112.

Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265.

Zeng, Y., Chen, J., Shang, W., and Chen, T.-H. P. (2019). Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, **24**(6), 3394–3434.

Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D., and Stumm, M. (2014). Lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 629–644. USENIX Association.

Zhou, J., Zhang, H., and Lo, D. (2012). Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24.

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., and Weiss, C. (2010). What makes a good bug report? *IEEE Transactions on Software Engineering*, **36**(5), 618–643.

Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? page 1–5.