An Exploratory Study of Logging Configuration Practice in Java

Chen Zhi*†, Jianwei Yin*, Shuiguang Deng*†, Maoxin Ye*, Min Fu‡§, Tao Xie¶||
*Zhejiang University, Hangzhou, China
†Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Hangzhou, China
‡Alibaba Group, Hangzhou, China
§Macquarie University, Sydney, Australia
¶Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
||University of Illinois at Urbana-Champaign, Urbana-Champaign, USA
{zjuzhichen, zjuyjw, dengsg, ymx}@zju.edu.cn, hanhao.fm@alibaba-inc.com, taoxie@illinois.edu

Abstract—**Logging components are an integral element of software systems. These logging components receive the logging requests generated by the logging code and process these requests according to logging configurations. Logging configurations play an important role on the functionality, performance, and reliability of logging. Although recent research has been conducted to understand and improve current practice on logging code, no existing research focuses on logging configurations. To fill this gap, we conduct an exploratory study on logging configuration practice of 10 open-source projects and 10 industrial projects written in Java in various sizes and domains. We quantitatively show how logging configurations are used with respect to logging management, storage, and formatting. We categorize and analyze the change history (1,213 revisions) of logging configurations to understand how the logging configurations evolve. Based on these study results, we reveal 10 findings about current practice of logging configurations. As a proof of concept, we develop a simple detector based on some of our findings. We apply our detector on three popular open-source projects and identify three long-lived issues (more than two years). All these issues are confirmed and two of them have been fixed by the open-source developers.**

Index Terms—**logging, logging configurations, empirical study**

I. Introduction

Logging is a common practice to record runtime information for realtime analysis and postmortem inspection. The rich information in logs is crucial for many tasks, such as failure diagnosis [1], [2], performance analysis [3], and user behavior analysis [4]. The generation of logs is implemented by a logging component in the software. The logging component receives logging requests from application code and publishes the logging information to the specified destinations. It is not trivial to build a well-designed logging component, considering the vast variety of logging requirements. Therefore, logging libraries, such as Log4J [5] and Logback [6] in Java, are adopted to build a logging component. These logging libraries not only simplify the writing of the logging component within the software, but also provide logging configurations, with the flexibility of controlling logging behaviors from an external configuration file.

Although the logging libraries have dramatically simplified the construction of a logging component, it is still challenging to write logging code for producing high-quality log messages. Previous studies find that many problems exist in the practice of writing logging code, such as missing failure information [7], improper logging level [8], and duplicated log message [9]. In addition, various approaches [9]–[15] have been proposed to improve current logging code.

Besides writing logging code, conducting logging configurations is also highly important for producing high-quality log messages; however, no existing research studies the practice of logging configurations. The data released by Hassani et al. [14] show that majority (65%) of the issues related to logging configurations in the Apache Hadoop project have medium or high priority. These issues are possible to cause critical problems, such as runtime errors [16], disk-space exhaustion [17], and log-message missing [18]. In contrast, among other logging-related issues in these projects, there are only about 39% of them with medium or high priority. Therefore, the issues in logging configurations are more likely (1.6 X) to bring risk to the logging component, or even the entire system.

To fill this gap of lacking studying logging configuration practice, in this paper, we conduct an exploratory study for aiming to answer the following two research questions.

RQ1: How are the logging configurations used? To answer this research question, we intend to analyze the logging configurations in some typical software systems. Specifically, we focus on the core elements in logging configurations, namely, logger, appender, and layout, being critical for logging management, logging storage, and logging formatting, respectively:

- Logger is responsible for capturing and managing logging requests.
- Appender is responsible for recording logging requests to a destination.
- Layout is responsible for converting and formatting the data in a logging request.

RQ2: How do the logging configurations evolve? This research question aims to analyze all possible changes in logging configurations to find out what change types usually occur in practice. In the meantime, it is helpful to explain common practice of logging configurations. Specifically, we examine all the valid changes to logging configurations and categorize them manually in term of changed elements (e.g., logger, appender, and layout).

This paper makes the following main contributions:

1) To the best of our knowledge, our work is the first dedicated to explore the practice of logging configurations by analyzing the usage and evolution of logging configurations from 20 (10 open-source and 10 industrial) software projects in various sizes and domains.
2) Based on analyzing the study results, we reveal 10 findings about logging configurations, including logging management, logging storage, logging formatting, and logging-configuration quality. Table I summarizes our major findings from our study.
3) Based on our findings, we implement a tool to detect invalid loggers in logging configurations. The tool has been applied to three popular open-source projects and identify three long-lived issues (more than two years), demonstrating the usefulness of our findings.

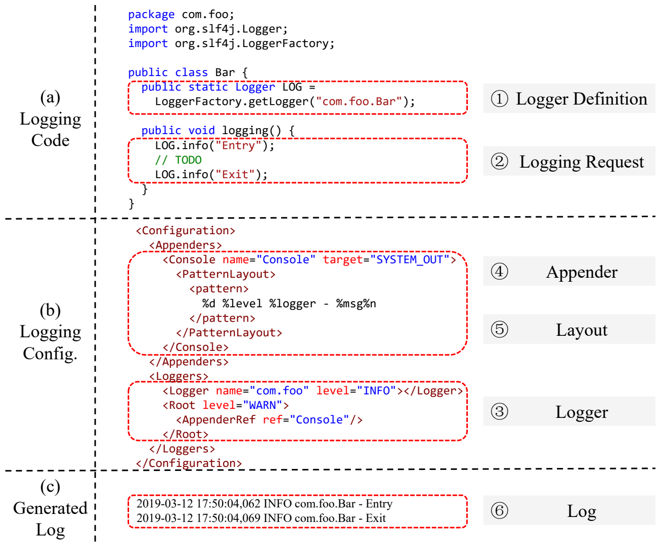
II. Background

In this section, we describe the logging mechanism adopted by common logging libraries. Figure 1 shows an illustrative example for the logging mechanism of Log4J 2 [5]. Specifically, Figure 1(a) shows the application code that contains logging code, Figure 1(b) shows the corresponding logging configurations in the `XML` format, and Figure 1(c) shows the generated logs by running the application code.

Before we can send logging requests to the logging component, we need to obtain and name a logger instance. In our case, the named logger is `com.foo.Bar`. The name of logger follows the **hierarchical naming** rule [6]: *A logger is said to be an ancestor of another logger if its name followed by a dot is a prefix of the descendant logger name. A logger is said to be the parent of a child logger if there are no ancestors between itself and the descendant logger.* For instance, `com.foo` is the parent of `com.foo.Bar`, while `com` is only the ancestor of `com.foo.Bar`. In addition, there is a root logger, which is an ancestor of any logger by default.

Then, we can send a logging request using the logging method with the developer-written logging level. The logging component compares the assigned logging level with the configured threshold for the logger to verify whether the logging request should be processed. Generally, there are six logging levels, which are *fatal*, *error*, *warn*, *info*, *debug*, and *trace* with decreasing priority. The level of logger follows the **level inheritance** rule [6]: *The level for a given logger L, is equal to the first configured level in its hierarchy, starting at L itself and proceeding upwards in the hierarchy towards the root logger.* In Figure 1(b), the first configured level for `com.foo.Bar` is *info*, which is inherited from `com.foo`. The developer-written level (*info*) is not lower than the configured threshold, so the logging requests are transferred to the associated appenders.

The appender of logger follows the **appender additivity** rule [6]: *The log message of logger L goes to all the appenders in L and its ancestors.* In our example, as `com.foo.Bar` does not bind with any appender directly, the log message of `com.foo.Bar` is directed to only *ConsoleAppender*, which is configured with the root logger. Before writing the log message to the destination, the logging component utilizes the associated layout to format the log message. Here, the *PatternLayout* is used to format the log message according to the value of `pattern`. Specifically, it adds the time, priority, and category into the log message. Finally, the formatted log message is outputted to the console.



```
package com.foo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Bar {
  public static Logger LOG =
    LoggerFactory.getLogger("com.foo.Bar");

  public void logging() {
    LOG.info("Entry");
    // TODO
    LOG.info("Exit");
  }
}
```

```
<Configuration>
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout>
        <pattern>
          %d %level %logger - %msg%n
        </pattern>
      </PatternLayout>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="com.foo" level="INFO"></Logger>
    <Root level="WARN">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

```
2019-03-12 17:50:04,062 INFO com.foo.Bar - Entry
2019-03-12 17:50:04,069 INFO com.foo.Bar - Exit
```

Fig. 1: An illustrative example of logging mechanism

III. Study Methodology

In this section, we first introduce the systems under study, and then describe how we extract necessary information for further analysis.

A. Subject Systems

We perform our study on 10 open-source projects and 10 industrial projects in Java. The open-source projects are selected from the Apache Software Foundation [21] based on three main criteria. First, the selected projects are popular with more than 1,000 stars on Github. Second, the selected projects are mature with more than eight years of development history. Third, the selected projects represent various domains, ranging from message queue to distributed computing platforms. The industrial projects are retrieved from the Alibaba Group, which is the world's largest retailer, and one of the largest Internet companies in the world. These projects are fundamental systems to support the business of the company. Half of them are core business platforms, such as product management, membership management, and fund management. The other half of them are middleware, such as cache server, message broker, and configuration server.

The details of these projects are listed in Tables II and III, respectively. The LOC (lines of code) measures the size of functional code, excluding the testing code, comments, and white-spaces. The LOLC (lines of logging configurations)

TABLE I: Major findings in our study

Logging Management (Findings 1, 2, 5, and 8)	Implications
Developers make efforts to take full advantage of the logging mechanism to simplify logging management, such as adopting a well-designed naming convention. However, there are some common cases that default behaviors of the logging mechanism result in unexpected logging behaviors (e.g., duplicated log messages).	It is necessary to establish conventions to consolidate the best logging practices. For example, there is a dedicated section for logging practice in Alibaba Java Coding Guidelines [19].
About one third (32.4%) of loggers are configured to control the logging activities of external libraries. There are some external libraries whose logging behaviors are often recognized as inappropriate.	It is possible to extract templates of logging configurations for some common external libraries, especially for common dependencies in software ecosystems.
Majority (80%) of the changes to thresholds are adjustments between different thresholds. It indicates that developers struggle to assign appropriate thresholds for loggers to strike a balance between different logging levels and between different logging activities.	It is not trivial to determine thresholds for loggers under the current logging mechanism. Recent work on new logging mechanisms (e.g. adaptive logging [20]) may address this problem.
Logging Storage (Findings 3 and 6)	Implications
Reliability and performance are the main concerns of logging storage. The most commonly used appender is rolling-file appender. More than one third (38.6%) of appender change the storage parameters at least once to meet ever-changing capacity and performance requirements.	It is not trivial to determine the storage parameters and there is a demand for automatic capacity estimation and performance tuning.
Logging Formatting (Findings 4 and 9)	Implications
A significant percentage (94.3%) of layouts add several (with a median of 3) useful contexts when the logging component formats the logging requests. More than half (65%) of them have been changed at least once to improve informativeness, performance, or understandability (both for human and machine) of log messages.	It is possible to add common context information into logging messages by customizing the layouts, instead of changing multiple instances of logging code.
Logging-Configuration Quality (Findings 7 and 10)	Implications
About 12.9% of changes to logging configurations are to improve their quality. Specifically, developers extract configurable variables from logging configurations to enhance usability and replace the names in logging configurations with more readable ones to strengthen maintainability.	Additional work and tools are needed to improve the quality of logging configurations, such as detecting and resolving the smells in logging configurations automatically.

TABLE II: Statistics of open-source projects

Project	Description	Version	SLOC	LOLC
ActiveMQ	Message broker	5.15.8	210K	108*
Ambari	Cluster management	2.7.3	285K	139*
Cassandra	Distributed database	3.11.3	284K	85
Flume	Log management	1.8.0	49K	91*
Hadoop	Distributed computing	2.9.2	717K	429*
HBase	Distributed database	2.1.1	381K	154*
Hive	Data warehouse	3.1.1	992K	113*
Solr	Search engine	7.5.0	634K	68
Storm	Distributed computing	1.2.2	181K	87
Zookeeper	Distributed coordinator	3.4.13	37K	69*
Total			4M	1,343

* The original configuration files are written in the `properties` format and we convert them into the equivalent `XML` format manually. Because XML-style files are more widely used than property-style ones, and many advanced features are supported in only XML-style files.

TABLE III: Statistics of industrial projects

Project	Description	SLOC	LOLC
P1	Business platform	445K	875
P2	Business platform	62K	1,233
P3	Business platform	276K	784
P4	Business platform	308K	993
P5	Business platform	333K	1,468
P6	Cache server	207K	387
P7	Configuration server	13K	246
P8	Database replication	61K	87
P9	Log management	78K	60
P10	Message broker	105K	416
Total		2M	6,549

stands for the size of logging-configuration files in the `XML` format, excluding empty lines but including comments, because we find many configurations are commented out in open-source projects to disable some functionalities by default. It is worth noting that the average size of logging configurations in open-source projects is smaller than that of industrial projects. We suspect that open-source projects mainly provide some common logging configurations during development, and users of these open-source projects can customize the logging configurations in production to meet various requirements, e.g., audition, operation.

B. Data Extraction

We next describe the steps that we use to extract the necessary information from these projects.

1) Identifying Logging Configuration Files: The first step is to find the logging-configuration files for these projects. Note that there are different configuration files in projects for various purposes (e.g., test configurations, sample configurations). In this study, we focus on the core production configuration files. We devise the following command to find all the configuration files:

```
find | grep -E ".*log.*\.(properties|xml)"
```

This command finds all the files whose name contains "log" and ends with `properties` or `XML`. Among the found files, we manually identify the core configuration files. After having found the logging-configuration files for each project, we extract details of configurations (e.g., logger) from these files to answer the first research question (Section IV). This process is automatic; we have built a simple script based on heuristic rules.

2) Extracting Logging-Configuration Revisions: In order to answer the second research question (Section V), we extract all the revisions to logging-configuration files. In particular, we retrieve the `git` repositories of these projects, and we use the following command to extract the commits that contain logging-configuration revisions:

```
git log -M --follow [filepath]
```

Note that this command does not work when the content of files is changed too much (e.g., file format change). To address this issue, we manually examine the first revision of the configuration files and determine whether it is derived from other files.

Once we obtain all the revisions, we filter the invalid revisions containing only changes that we are not interested in. These invalid revisions mainly fall into the following types:

- **Revisions do not change the content of logging configurations**. The addition, deletion, and movement of logging-configuration files belong to this type. In addition, the revisions changing only the whitespace in logging-configuration files are also ignored.
- **Temporary revisions**. The revisions belonging to this type often come in pair. The first revision introduces some changes for temporary tasks, such as testing and debugging. Once the tasks are finished, developers commit the other revision to revert the changes of the first revision.
- **Duplicate revisions**. This situation often occurs when developers use `cherry-picking` to apply some revisions from one branch to another branch and then merge these two branches into the same branch.
- **Rollback revisions**. This type of revisions is mainly caused by carelessness of developers who forget the changes to logging configurations when merging different branches. The developers sometimes make another complementary revision to get those missing changes back.

C. *Categorizing Logging-Configuration Changes*

Actually, developers often commit unrelated or loosely related changes in a single revision [22]. Therefore, the changes to logging configurations in each revision consist of some atomic changes. We first divide the changes into atomic changes according to their semantics. After that, we examine each revision based on its commit message and detailed patch. If the revision is associated with issue reports in the issue tracking system (e.g., JIRA), we also investigate the description, discussion, and other information stored in every associated issue report to understand the revision properly. We further categorize the atomic changes according to the changed elements, which can reflect the semantics of the changes. This part of work is conducted by the first and fourth authors jointly. The two authors have categorized the revisions independently and then discussed the disagreements to reach consensus.

IV. RQ1: How are the logging configurations used?

As explained in Section II, logging configurations are mainly composed of loggers, appenders, and layouts. To answer this research question (RQ1), we investigate four key aspects of these core elements: the name and source of logger, the type of appender, and the pattern of layout.

- The name of logger influences how efficiently we can manage the logging requests.
- The source of logger affects what kind of logging requests we need to manage.
- The type of appender determines the storage process of log messages.
- The pattern of layout impacts how the log messages are formatted.

TABLE IV: Details of loggers

Project	Naming Conventions			Source		
	Package	Topic	Mixed	Internal	External	Unknown
ActiveMQ	6	0	1	2	5	0
Ambari	8	4	0	9	3	0
Cassandra	1	0	0	1	0	0
Flume	6	0	0	1	5	0
Hadoop	4	3	2	7	2	0
HBase	7	0	1	2	6	0
Hive	24	4	0	4	24	0
Solr	3	0	1	2	2	0
Storm	3	0	0	3	0	0
Zookeeper	0	0	0	0	0	0
Subtotal	62	11	5	31	47	0
Percentage	79.5%	14.1%	0.06%	39.7%	60.3%	0.0%
P1	24	38	1	47	14	2
P2	19	31	4	10	44	0
P3	16	38	1	38	10	7
P4	34	81	0	86	28	1
P5	12	51	0	51	12	0
P6	6	7	0	13	0	0
P7	0	0	11	11	0	0
P8	0	3	0	3	0	0
P9	0	4	0	4	0	0
P10	0	19	0	18	0	1
Subtotal	111	272	17	281	108	11
Percentage	27.8%	68.0%	4.3%	70.3%	27.0%	2.8%
Total	173	283	22	312	155	11
Percentage	36.2%	59.2%	4.6%	65.3%	32.4%	2.3%

A. *Loggers*

Loggers are also known as categories, and they are the entry point to the logging component and are responsible for capturing logging requests and redirecting them to the appropriate appenders.

1) Naming conventions of loggers: All the loggers need to be named before they can be used. The names of loggers are critical because all the logger are organized in an inheritance relationship, as explained in Section II. Benefiting from the inheritance relationship, we need to manage only a limited number of loggers, and others inherit the configurations from their nearest configured ancestor loggers. Therefore, it is important to follow a well-designed naming convention.

The most commonly used naming convention is package-based naming. As shown in Figure 1, when using this naming convention, the loggers are named with the package names or class names. However, doing so faces an issue when multiple distinct logging activities need to be performed under the same package or class. Topic-based naming can be used to address this issue, because topic-based naming names the logger according to the topic of logging activities. To analyze the usage of the naming convention, we write a simple script to extract all logger names from the logging configurations in the projects under study, and then recognize what naming conventions are followed by each logger name using heuristic rules. In the rest of this paper, "package-based loggers" and "topic-based loggers" refer to loggers that are named following the conventions of package-based naming and topic-based naming, respectively.

As a result, Columns 2-4 in Table IV show the distribution of loggers according to naming conventions across projects. The results show that majority (79.5%) of the loggers in configurations of open-source projects follow the convention of package-based naming. In addition, the number of package-based loggers is larger than the number of topic-based loggers in each open-source project. However, the results of the industrial projects are very different from those of the open-source projects. In the industrial projects, the topic-based loggers account for the largest proportion (68%). One possible reason is that industrial projects prefer the flexibility of logging configurations because of various logging requirements. For example, we have observed that some classes in P5 perform more than five distinct logging activities.

Note that we find some special loggers whose name starts with the package name and ends with a topic. As shown by Column 4 (denoted with "Mixed") of Table IV, this special naming convention appears in 8 projects. Diving into the details of these loggers, we find out two situations: (1) the separate topic-based loggers are dedicated to specific package-based contexts; (2) some topic-based loggers share the same package-based contexts. In both situations, developers intend to control these topic-based loggers together with other loggers under the same contexts. Hence, the developers adopt this particular naming convention. For example, issue YARN-6042 from issue repositories (i.e., JIRA) introduces a separate logger to print state dump of fair scheduler and it is named with the class name of fair scheduler following a constant string ".statedump".

Finding 1: There are three naming conventions adopted by developers, namely, topic-based naming (59.2%), package-based naming (36.2%) and mixed naming (4.6%), as shown in the last row of Table IV.

2) Sources of loggers: Software projects rarely work in isolation. In most cases, a project relies on reusable functionalities in other libraries. Due to the extensive usage of external libraries, many pieces of logging code belonging to these external libraries are also imported into the host project, and some of these code pieces are invoked during the runtime. In order to avoid undesirable logging behaviors from external libraries, developers sometimes control the logging activities by configuring these external libraries' loggers, which we refer to as "external loggers". Similarly, we refer to the loggers defined in the host project as "internal loggers".

To identify the sources of loggers in configurations, we find out where these loggers are defined. To this end, we build a tool to extract all the logger definitions in code. Then we iterate through each logger in configurations to find the matched loggers in code. For those unmatched loggers, we infer their sources using heuristic rules.

Consequently, Columns 5-7 of Table IV show the distribution of loggers with respect to the sources across projects. Note that there are some loggers whose source cannot be determined, even if we use heuristic rules. These loggers follow the convention of topic-based naming, but the topics (e.g., "MONITOR", "SQL") are too general to determine which libraries they belong to.

External loggers are found in 12 out of 20 projects. Surprisingly, 5 out of these 12 projects have external loggers no fewer than internal loggers. For example, the number of external loggers is 6 times that of internal loggers in *Hive*. *Hive* is a data warehouse system built on the *Hadoop* ecosystem, and there are 15 out of 24 external loggers from the *Hadoop* ecosystem. Moreover, we observe that there are some common external loggers that are configured to restrict their logging activities, implying that the logging behaviors of the corresponding external libraries are often recognized as inappropriate. For example, *Solr*, *Hive*, and *HBase* have the same external loggers about *Zookeeper* in their logging configurations. It is possible to suggest some logging configurations when new external libraries are introduced.

Finding 2: About one third (32.4%) of loggers are configured to control the logging activities of external libraries, as shown in the last row of Table IV. There are some external libraries whose logging behaviors are often recognized as inappropriate.

TABLE V: Distribution of appender types

Type	Subtype	Freq.	Ratio
File	Size based rolling	123	28.6%
	Size and time based rolling	101	23.5%
	Time based rolling	90	20.9%
	No rolling	5	1.2%
Subtotal		319	74.2%
Asynchronous	N/A	58	13.5%
Console	N/A	13	3.0%
Other	N/A	40	9.3%
Total		430	100.0%

B. Appenders

Appenders are also known as handlers, and they are responsible for recording logging requests to the destinations. Logging libraries provide a vast range of appenders (e.g., file, database) to meet different output requirements. We have extracted all appenders in logging configurations from the projects under study and Table V shows the distribution of the appender types.

It is indeed expected that file appenders are the most common appenders (74.2%), because most of the log data are stored as files on storage devices. The second most common appenders are asynchronous appenders, which are able to process logging requests asynchronously and often introduced to improve logging performance of existing appenders. Console appenders are in the third place. In some cases, the console is the only available destination to display log messages. Most of the rest are customized appenders, and they are mainly used to meet special storage requirements for certain log messages.

For file appenders, we further divide them into subtypes in term of rolling policies. As the log files tend to grow over time, they become unmanageable and bring the risk of crashing. Rolling policies are responsible to address this problem by archiving log files when certain predefined conditions are satisfied. As shown in Table V, there are only 1.6% (5/319) file appenders without rolling. 4 of them come from *Ambari*, for which the frequency of logging requests is low (e.g., infrequent

TABLE VI: Distribution of conversion characters

Characters	Freq.	Brief Description
%n	346	line separator character
%msg	334	developers supplied message
%date	313	the date (time, time zone, etc.)
%mdc	246	developers supplied thread-context message
%level	224	developers supplied logging level
%logger	190	developers supplied logger name
%ex	42	control the depth of stack trace
%thread	36	the name of the thread
%line	13	the line number in source file
%class	6	the fully qualified class name of the caller
Other	9	such as the method of the caller

user behavior) or constant (e.g., environment checking during system initialization). The last one is from *Zookeeper*, which is designed to store trace logs during debugging.

The most commonly used rolling policy is size based rolling, which accounts for 38.6% (123/319) of all file appenders. It rolls log files over when their sizes reach predefined thresholds. Time based rolling takes the third place among all rolling polices. With the help of time based rolling, it is possible to roll log files over hourly or daily, being helpful for developers to narrow down the search space and locate the relevant log messages based on time ranges. In addition, size and time based rolling has the second highest proportion among all the rolling polices. It rolls log files over according to time and file sizes, and is devised to take advantage of strengths from size-based rolling and time-based rolling.

Finding 3: The majority (74.2%) of appenders are file appenders, and almost all (98.4%) of them are configured with rolling policies, as shown in Table V.

C. Layouts

Layouts are also known as formatters, and they are responsible for converting and formatting the data in logging requests. In the projects under study, all the appenders associated with layout adopt *PatternLayout* (334) to format the logging message. *PatternLayout* formats the logging information using a given pattern string, which is composed of literal texts and conversion characters. Logging libraries provide many built-in conversion characters to display commonly used data, e.g., date, priority. Table VI summarizes the distribution of conversion characters in the projects under study.

Note that we exclude %n, %msg, and %ex from our discussion. %n and %ex are merely used to control the format and they do not generate any information. %msg is the message generated by developer-written logging code and all layouts include this conversion actually. In addition, we find that there are 315 (94.3%) layouts that contain other (with a median of 3) informative conversions except these 3 conversions.

Finding 4: A significant percentage (94.3%) of layouts add several (with a median of 3) useful contexts when logging components format the logging message.

V. RQ2: How do the logging configurations evolve?

In this section, we investigate how the logging configurations evolve. Following the steps described in Section

TABLE VII: Revision of projects under study

Project	# Total Rev.	# Total LC Rev.	# Valid LC Rev.	% Valid / Total LC Rev.
ActiveMQ	10K	26	24	92.3%
Ambari	24K	19	17	89.5%
Cassandra	23K	41	30	73.2%
Flume	2K	12	11	91.7%
Hadoop	15K	41	37	90.2%
HBase	16K	37	30	81.1%
Hive	12K	32	27	84.4%
Solr	30K	22	20	90.9%
Storm	8K	32	29	90.6%
Zookeeper	1K	9	7	77.8%
Subtotal	140K	271	232	85.6%
P1	16K	59	53	89.8%
P2	44K	108	88	81.5%
P3	19K	144	110	76.4%
P4	23K	160	149	93.1%
P5	44K	144	131	91.0%
P6	5K	53	26	49.1%
P7	1K	19	15	78.9%
P8	2K	23	16	69.6%
P9	2K	35	26	74.3%
P10	2K	58	42	72.4%
Subtotal	158K	803	656	81.7%

[*] LC is abbreviation for "logging configurations".

TABLE VIII: Description of change types

Type	Brief description
Schema	Add/delete loggers, appenders and association between them
Storage	Change the polices and parameters of appenders
Threshold	Change the thresholds of loggers and appenders
Layout	Change the patterns of layouts
Name	Change the (logger, appender, etc.) names in configurations
Variable	Change the variables in configurations

III-B2, we have extracted all revisions related to logging configurations, and Table VII shows the summary statistics of these revisions for each project under study. Column 3 lists the numbers of revisions related to logging configurations, while Column 4 lists the numbers of valid revisions related to logging configurations. Column 5 shows that the ratio of valid over total revisions related to logging configurations mostly fall into 70-90%. Note that the ratio of *P6* is much lower because there are many temporary changes for debugging. In summary, we have examined 1,213 revisions that contain changes to logging configurations, and 891 of them turn out to be valid revisions.

Subsequently, we follow the instructions introduced in Section III-C to categorize the atomic changes. As a result, we have found 12 change types in term of changed elements. Due to the limited space, we discuss only the top half of them[1], which account for 93.2% of all changes to logging configurations. The description and distribution of these change types are presented in Tables VIII and IX, respectively. We can see that the most common change types are the same in open-source and industrial projects, but with slight different orders.

Figure 2 illustrates the distribution of priority for change types. The priority is extracted from the associated issue reports of open-source projects. The order of stacks in Figure

[1]The complete list of these change types can be found in our complementary materials: https://github.com/log-config/log-config

TABLE IX: Distribution of changes types

Industrial Projetcs			Open-source Projects		
Type	Freq.	Ratio	Type	Freq.	Ratio
Schema	433	48.2%	Schema	97	30.8%
Threshold	151	16.8%	Threshold	57	18.1%
Storage	107	11.9%	Layout	33	10.5%
Name	76	8.5%	Name	30	9.5%
Layout	72	8.0%	Storage	24	7.6%
Variable	27	3.0%	Variable	23	7.3%
Other	32	3.6%	Other	51	16.2%
Total	898	100.0%	Total	315	100.0%

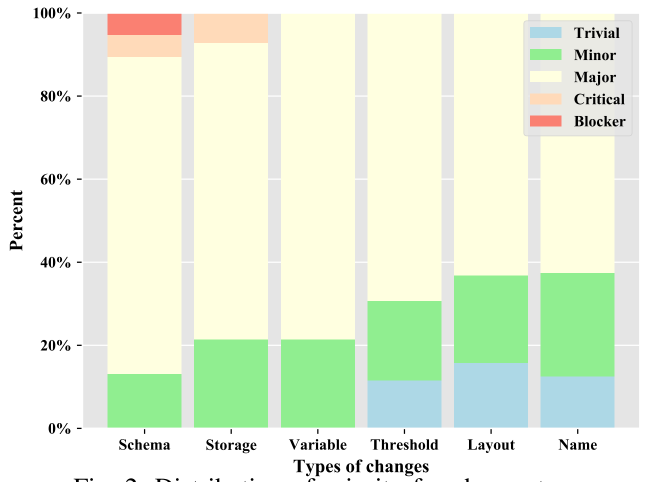


Fig. 2: Distribution of priority for change types

2 is derived from the proportion of issue reports with medium or high priority (i.e., labeled as Major, Critical, or Blocker) for corresponding change types. In the rest of this section, we also follow the order to describe the change types.

A. Schema Changes

We use the term “schema” to represent the logical structure of logging configurations; schema is determined by loggers, appenders, and association between them. The changes to schema have the highest proportion and priority among all change types. According to what elements have been changed, we divide this category into the following two subtypes.

Addition/deletion of loggers or appenders (89.6%). This type of changes accounts for the majority of schema changes. More than one third of these updates (i.e., addition or deletion) are adaptive changes, which often appear as a revision that contains updates of feature code and corresponding logging configurations. For the remaining independent updates, there are two main causes: (1) separating specific log messages from others to facilitate post-processing; (2) changing the thresholds for certain log messages.

Modifications to association between loggers and appenders (10.4%). The association between loggers and appenders can be explicit or implicit. The explicit associations are specified by the *AppenderRef* attribute of logger in configuration. Developers change the explicitly associated appenders of loggers to change the logging behaviors. In the projects under study, half of these changes are applied to the root logger to switch between the production-logging mode and development-logging mode; the other half of them are used to switch between synchronous appenders and asynchronous appenders for performance improvement.

TABLE X: Migration of rolling policies

Migration Path	Freq.
Time based rolling $\rightarrow$ Size based rolling	84
Time based rolling $\rightarrow$ Size and time based rolling	54
No rolling $\rightarrow$ Time based rolling $\rightarrow$ Size based	3
No rolling $\rightarrow$ Time based rolling $\rightarrow$ Size and time based rolling	3
Size based rolling $\rightarrow$ Time based rolling	2
No rolling $\rightarrow$ Size based rolling	1

The implicit associations are derived from appender additivity of logging mechanism. Developers sometimes turn off appender additivity to eliminate undesirable logging behaviors, such as duplicated log messages (e.g., issue HIVE-11563) and cramming unexpected appenders (e.g., issue YARN-6360). Specifically, we notice that if the logger is configured with explicit association, its appender additivity is likely (97.1%) to be turned off in the projects under study.

Finding 5: Almost all (97.1%) loggers configured with explicit association remove implicit associations to avoid undesirable logging behaviors, caused by the default appender additivity.

B. Storage Changes

This category refers to those changes that edit the details of existing appenders to adjust storage behaviors. According to the effects of the changes, this category can be divided into two cases: policies adjustment and parameters tuning.

Policies adjustment (42%). This subtype refers to the changes of appender types, especially for the rolling policies. We have extracted the change history for each appender with respect to rolling policies. As a result, Table X shows the migration paths of rolling policies for each appender. We can see that the majority of the migrations occur to time based rolling policy. The migrations are mainly performed to improve the reliability of logging storage, because some implementations of time based rolling policy do not support to limit the total size of log files, and the generated logs have the risk to fill up storage devices. For example, issue HADOOP-8149 asks to change the default rolling policies from time based rolling to size based rolling. The issue reporter states “*I’ve seen several critical production issues because logs are not automatically removed after some time and accumulate.*”

Parameters tuning (58%). Appenders occasionally offer some configuration parameters to control their detailed storage processes. We have tracked the changes to parameters for each appender and find that more than one third (38.6%) of the appenders have changed their parameters at least once (with median of 2). These parameters can be simply classified into the following two types:

- Capacity-related parameters. These parameters are mainly adopted by rolling polices to limit the size of individual log file or total log files. Developers usually need to adjust these parameters to meet the varied capacity requirements. Moreover, developers sometimes need to balance the capacities assigned to each appender based on their priorities.

- Performance-related parameters. Performance is one of the main concerns about logging. Developers usually enable some advanced features (e.g., buffering and asynchronization) to improve logging performance. However, we find that some changes are to turn off these features, as they cause some unexpected side effects. For example, we observe that some appenders used to output realtime-monitoring logs have turned off buffering to assure the immediacy of log messages (e.g., issue STORM-1519).

Finding 6: Reliability and performance are the main concerns of logging storage. More than one third (38.6%) of appenders change the storage parameters at least once to meet ever-changing capacity and performance requirements.

C. Variable Changes

A variable can be specified in a logging configuration file and it is replaced with real values during the initialization of a system. This feature, often named *variable substitution*, is useful for automatic integration and deployment, especially in the current development practice. As there are multiple deployment environments, some properties (e.g., storage path) in logging configurations should be consistent with the environments. Actually, we find that all the projects under study have introduced at least one variable (with median of 3) into their logging configurations.

Most (70%) of these variable changes are on extracting variables, such as storage parameters and logging levels. These variables are mostly extracted for two reasons. The first is to replace values that are varied according to context. The variables for storage parameters belong to this type. The second is to replace values that are often changed simultaneous. Such situation often occurs to logging levels of appenders, because their logging activities share some commonality with each other and their logging levels should be kept consistent with each other. The remaining variables changes are switching between variables and hard-coded values. Such situation usually occurs to where particular values need to be used.

Finding 7: There are at least one variable (with median of 3) defined in logging configurations of the projects under study. And the majority (70%) of the changes to variables are to introduce new variables.

D. Threshold Changes

Loggers and appenders can have thresholds associated with them. Logging requests are filtered if their levels are lower than thresholds of both the loggers and appenders. Hence, it is possible to turn off or on certain levels of logging by changing the thresholds attached to the loggers and appenders. Threshold changes account for 17.1% of the changes to logging configurations, being in the second place after schema changes. Note that there are rare cases of changing the threshold of appenders, and we focus on only those changes to loggers. We further break down threshold changes into the following three subtypes.

Increasing threshold (55.9%). These changes are performed to filter log messages, and these changes occur to external loggers and internal loggers evenly. There are two main cases for these changes: (1) the thresholds are changed from *debug* to *info*, as the corresponding software components are sufficiently stable for production; (2) the thresholds are changed from *info* to *warn* or *error*, as the corresponding loggers generate excessive trivial log messages and affect the efficiency for diagnosis.

Decreasing threshold (24.1%). Developers sometimes relax the thresholds of loggers to allow more log messages for diagnosis. Majority (76.1%) of these changes occur to internal loggers. Interestingly, we find some cases where developers initially increase the thresholds of certain loggers to filter out some noisy log messages (e.g., issue HBASE-1273), but finally find that the thresholds are too high, causing to lose some important log messages, so they decrease the thresholds (e.g., issue HBASE-1572). Such result indicates that developers struggle to assign appropriate thresholds for loggers to balance different logging activities.

Breaking level inheritance (20%). Developers sometimes explicitly assign loggers with the thresholds of their parent loggers to avoid level inheritance. These changes are usually used to restrict the log messages generated by these loggers during debugging. In such situation, the parents of these loggers usually are the root loggers, whose thresholds are decreased during debugging (e.g., issue FLUME-1418).

Finding 8: Majority (80%) of the changes to thresholds are adjustments between different thresholds. It indicates that developers struggle to assign appropriate thresholds for loggers.

E. Layout Changes

Developers can specify pattern layouts to format log messages. We have extracted all the changes for each layout and found that most (65%) of them have been changed at least once. In term of editing actions, this category can be divided into the following three subtypes.

Layout enhancement (61.9%) is to add extra information into the layouts. The three most commonly added conversions are %mdc, %date, and %thread. %mdc and %thread belong to thread contexts, being usually helpful to debug multi-threaded applications. %date provides a flexible definition of date format, and developers adjust the format according to their needs.

Layout simplification (26.7%) is to remove some information from the layouts. More than half of the simplifications are to remove useless information. For example, in revision `b0269719`, *ActiveMQ* removes the logger names for audit log. The reason is that it is easy to identify the category or locate the related classes for audit logs without the logger names. One third of the simplifications are to remove redundant information. For instance, in revision `b0269719`, *ActiveMQ* removes the date for audit logs, as the date has been printed in logging code. The remaining simplifications are dedicated to simplify location related information (e.g. %line, %class). As the generation of location information is extremely slow, the generation is avoided if performance is the main concern. For example, in issue HIVE-14079, the line number from pattern layout is removed.

Layout normalization (11.4%) does not change the contents of layouts, but changes the way to display the information, such as adding separators between conversions, adjusting the order of conversions and limiting the length of conversions. This subtype is mainly for ease of log parsing, because it is quite difficult to parse arbitrary log messages automatically.

Finding 9: More than half (65%) of layouts have been changed at least once to improve informativeness, performance, or understandability (both for human and machine) of log messages.

F. Name Changes

Similar with source code, developers also need to name entities in logging configurations, e.g., logger names, appender names, and variable names. We break down these changes to the following three subtypes according to their change purposes.

Coupling change (19.8%). There are some coupling relationships in logging configurations, especially for the couplings between logger definitions in code and logger usages in configurations. For these coupling names, the same changes should be applied for each of them.

Readability improvement (55.7%). This subtype of changes usually improves the readability of names or maintains the convention among names. For example, YARN-6453 changes the name of FairScheduler's appender from `FSLOGGER` to `FSSTATEDUMP`, because the original name is misleading, being amenable to be misinterpreted as "logger of file system".

Inconsistency repair (24.5%). This subtype can be divided into two cases: fixing copy-paste and fixing coupling changes. For the first case, developers add new loggers or appenders by copying existing configurations and applying similar edits to some locations. However, the developers sometimes introduce problems due to missing of some edits. For example, in revision `96c51312a`, *Storm* introduces a new appender "ACCESS" by copying the configurations of existing appender "A1". The developers change the name of log file, but forget to change the name of rolling log file. Hence, the rolling log files generated by "ACCESS" overwrite those generated by "A1". After half a year, in revision `7d4d1608f`, this issue is found and fixed. For the second case, it is due to missing of coupling changes. For example, in issue HADOOP-3951, the logger name of "FSNamesystem" is changed in code, but the corresponding logger in the log configuration is forgotten to be changed. After a year and a half, in issue HADOOP-7053, a complementary revision is made to fix this missing coupling change.

Finding 10: Maintainability is the main driver for changes to names in logging configurations. More than half (55.7%) of changes to names are to improve the readability by using well-defined names. One fourth of changes to names are fixing issues caused by inconsistency, and these issues are relatively hard to find, and they remain in logging configurations for years.

VI. Detection of Invalid Loggers

To demonstrate the usefulness of our findings, we have built a simple tool to detect invalid loggers based on our findings (Finding 10). We refer to *invalid loggers* as loggers that are used in configurations, but do not have any matched definition in source code. The reasons for these invalid loggers are that (1) the corresponding source code has been removed; (2) the logger definition in source code has been changed.

To this end, we have developed a static analysis tool based on WALA [23] from IBM to detect these invalid loggers. First, we scan the configuration files to get all the names of loggers used in configurations. Second, we retrieve all the class files (including those class files from external libraries) for each project and parse the class files to get all the names of loggers defined in code. Finally, we compare the two sets of loggers and report the unmatched loggers.

We apply our tool to the ten open-source projects under study (Table II), and our tool detects three issues in three open-source projects, as shown in Table XI. Note that the two unmatched loggers of *Hive* belong to the same issue, as they are introduced by the same external library. In addition, there is one false positive in *Hadoop*. The name of this logger is composed of a constant and a variable, and we have to implement constant propagation to determine the value of the variable; constant propagation is an expensive operation for such large-scale system. We report these issues to the developers of these three open-source projects, and these issues are all confirmed and two of them have been fixed. Actually, we notice that there are some perfective changes made to these invalid loggers after the missing coupling changes. However, the developers do not realize that these loggers are inconsistent with their definitions in source code. We also apply our tool to the ten industrial projects under study, and find some potential issues. We are working with the developers of these industrial projects to review these issues.

VII. Threats to Validation

Construct Validity. In our study, we consider only the logging configurations in files. However, logging libraries usually support to configure logging behaviors programmatically. These configurations are excluded from our study. Actually, programmatic configurations are rarely used because they are inflexible to change or manage. In addition, we do not take cloud logging into consideration, which is an emerging style of logging. The providers (e.g., Loggly [24] and Sumo Logic [25]) provide logging services in a cloud setting. However, there are no standard logging configuration models for this new style of logging.

TABLE XI: Result of applying our tool for detecting invalid loggers

Project	Source	Naming Convention	Logger Name	Status	Since
Hadoop	Internal	Mix	http.requests.s3gateway	False Positive	N/A
Hadoop	Internal	Package	org.apache.hadoop.mapred.JobInProgress$JobSummary	Fixed	2012-11-09
Ambari	External	Package	org.glassfish.jersey	Fixed	2016-04-28
Hive	External	Topic	JPOX	Confirmed	2015-12-02
Hive	External	Topic	Datastore	Confirmed	2015-12-02

Internal Validity. When we extract the revisions from the `git` repositories for each project under study, we find that the early development history is missing for some projects. Because these projects use other version control systems (e.g., `svn`) at first and migrate to `git` later. Some of these projects are relatively mature when the migration happens. Such factor is the reason why some large projects (e.g., P6) have a relatively small number of changes to logging configurations. If the early development history is included, there can be more valid revisions to logging configurations. However, we are inclined to believe that doing so influences the distribution of only existing types of changes and does not introduce new types of changes.

External Validity. We have only studied a limited number of projects in Java, and thus our results may not necessarily generalize to other projects. However, our projects under study cover a wide variety of projects of different sizes and domains. Moreover, the logging libraries in other programming languages (e.g., C#, Python) have borrowed many concepts from those logging libraries in Java, and are similar to them in terms of logging mechanisms.

VIII. Related Work

Our study is related to three categories of previous research: empirical studies on logging practice, empirical studies on configuration issues, and logging improvement. Different from the previous research, we focus on the practice of logging configurations, being also important but receiving insufficient attentions.

A. Empirical Studies on Logging Practice

Given the popularity and criticality of logging, previous research has been conducted to understand current logging practice. Yuan et al. [8] perform the first empirical study on logging practice in 4 open-source C/C++ software applications. They find that current logging practice is not good enough as developers are constantly making revisions to improve the quality of logging code. Chen et al. [26] and Zeng et al. [27] replicate the work of Yuan et al. [8] on Java software applications and Android applications, respectively. Some studies on industrial software [28] [29] also show that there is no rigorous logging specification even in a leading software company (e.g., Microsoft), and the logging behavior is highly developer dependent. Some other research focuses on certain aspects of logging practice, such as logging levels [30], logging-library migrations [31], description text in logging code [32], and energy consumption of logging code [33].

B. Empirical Studies on Configuration Issues

The issues in configurations have attracted research interests recently. Xu et al. [34] study the over-design issues in configurations and propose some techniques to simplify the design space of configurations. Xu et al. [35] analyze issues related to security-related configurations to understand the reasons for these security misconfigurations. Additionally, there exist studies to explore the practice of specific types of configurations, such as automatic-deployment configurations [36] and continuous-integration configurations [37].

C. Logging Improvement

Given that there are many issues in current logging practice, researchers also make efforts to help developers improve the quality of logging code. Previous research focuses on three categories. First, *where-to-log* addresses the problem of logging-code placement, and various techniques have been introduced to fulfill this goal based on program analysis [7] [10], machine learning [38], and association rule mining [11]. Second, *what-to-log* resolves the problem of insufficient information in logging code to improve the diagnosability of generated log messages [12]. Third, *log-repair* aims to detect and solve issues in logging code automatically, such as smells of duplicated logging code [9].

IX. Conclusion

In this paper, we have presented the first attempt to explore the practice of logging configurations using 10 open-source projects and 10 industrial projects written in Java in various sizes and domains. By analyzing the usage and evolution of logging configurations from these projects, we have revealed 10 findings about practice of logging configurations, including logging management, logging storage, logging formatting, and logging-configuration quality. We believe that these findings can inspire further work on improving the practice of logging configurations. Such benefits of our findings are confirmed by a simple tool built by us, motivated by the difficulties to identify certain invalid configurations. This tool has detected three long-lived issues (more than two years) from three popular open-source projects. All these issues have been confirmed, and two of them have been fixed by the open-source developers.

X. Acknowledgment

This work is supported by the National Key Research and Development Program of China (No.2017YFB1400603), National Natural Science Foundation of China under Grant (No.61825205 and No.61772459), National Science and Technology Major Project of China (No.50-D36B02-9002-16/19) and the Alibaba-ZJU Joint Research Institute of Frontier Technologies, NSF under grants no. CNS-1513939, CNS-1564274, and CCF-1816615, and grants from Futurewei and the ZJUI Research Program.

References

[1] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of 2009 ACM SIGOPS Symposium on Operating Systems Principles*, 2009, pp. 117–132.

[2] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 60–70.

[3] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proceedings of 2012 International Conference on Software Engineering*, 2012, pp. 145–155.

[4] S. Zhao, J. Ramos, J. Tao, Z. Jiang, S. Li, Z. Wu, G. Pan, and A. K. Dey, "Discovering different kinds of smartphone users through their application usage behaviors," in *Proceedings of 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2016, pp. 498–509.

[5] "Apache Log4j 2 manual," Accessed: 2018-08-01. [Online]. Available: https://logging.apache.org/log4j/2.x/manual/layouts.html

[6] "The logback manual," Accessed: 2018-08-01. [Online]. Available: https://logback.qos.ch/manual/architecture.html

[7] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: enhancing failure diagnosis with proactive logging," in *Proceedings of 2012 USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 293–306.

[8] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of 2012 IEEE/ACM International Conference on Software Engineering*, 2012, pp. 102–112.

[9] Z. Li, T.-H. P. Chen, J. Yang, and W. Shang, "DLFinder: characterizing and detecting duplicate logging code smells," in *Proceedings of 2019 International Conference on Software Engineering*, 2019, pp. 152–163.

[10] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of 2017 Symposium on Operating Systems Principles*, 2017, pp. 565–581.

[11] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "SMARTLOG: Place error log statement by deep understanding of log intention," in *Proceedings of 2018 IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 61–71.

[12] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems*, vol. 30, no. 1, pp. 1–28, 2012.

[13] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of 2017 International Conference on Software Engineering*, 2017, pp. 71–81.

[14] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3248–3280, 2018.

[15] S. Li, X. Niu, Z. Jia, J. Wang, H. He, and T. Wang, "LogTracker: learning log revision behaviors proactively from software evolution history," in *Proceedings of 2018 Conference on Program Comprehension*, 2018, pp. 178–188.

[16] "Enabling app summary logs causes FileNotFound errors," Accessed: 2018-08-01. [Online]. Available: https://issues.apache.org/jira/browse/YARN-812

[17] "Cap space usage of default Log4j rolling policy," Accessed: 2018-08-01. [Online]. Available: https://issues.apache.org/jira/browse/HADOOP-8149

[18] "Syslog missing from Map/Reduce tasks," Accessed: 2018-08-01. [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-5148

[19] "Alibaba Java coding guidelines," Accessed: 2018-08-01. [Online]. Available: https://alibaba.github.io/Alibaba-Java-Coding-Guidelines

[20] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *Proceedings of 2015 USENIX Annual Technical Conference*, 2015, pp. 139–150.

[21] "Apache Software Foundation," Accessed: 2018-08-01. [Online]. Available: https://projects.apache.org/projects.html

[22] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of 2013 Working Conference on Mining Software Repositories*, 2013, pp. 121–130.

[23] "T. J. Watson libraries for analysis (WALA)," Accessed: 2018-08-01. [Online]. Available: https://github.com/wala/WALA

[24] "Loggly," Accessed: 2018-08-01. [Online]. Available: https://www.loggly.com/

[25] "Sumo Logic," Accessed: 2018-08-01. [Online]. Available: https://www.sumologic.com/

[26] B. Chen and Z. M. Jiang, "Characterizing logging practices in Java-based open source software projects: a replication study in Apache Software Foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 1–45, 2016.

[27] Y. Zeng, J. Chen, W. Shang, and T.-H. P. Chen, "Studying the characteristics of logging practices in mobile apps: a case study on F-Droid," *Empirical Software Engineering*, pp. 1–41, 2019.

[28] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of 2014 International Conference on Software Engineering*, 2014, pp. 24–33.

[29] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: assessment of a critical software development process," in *Proceedings of 2015 IEEE/ACM International Conference on Software Engineering*, 2015, pp. 169–178.

[30] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1–33, 2016.

[31] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: a case study for the Apache Software Foundation projects," in *Proceedings of 2016 International Conference on Mining Software Repositories*, 2016, pp. 154–164.

[32] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of 2018 ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 178–189.

[33] S. Chowdhury, S. Di Nardo, A. Hindle, and Z. M. J. Jiang, "An exploratory study on assessing the energy impact of logging on Android applications," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1422–1456, 2017.

[34] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software," in *Proceedings of 2015 Joint Meeting on Foundations of Software Engineering*, 2015, pp. 307–319.

[35] T. Xu, H. M. Naing, L. Lu, and Y. Zhou, "How do system administrators resolve access-denied issues in the real world?" in *Proceedings of 2017 CHI Conference on Human Factors in Computing Systems*, 2017, pp. 348–361.

[36] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of 2016 International Conference on Mining Software Repositories*, 2016, pp. 189–200.

[37] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: an empirical study of projects that (mis)use Travis CI," *IEEE Transactions on Software Engineering*, p. 1, 2018.

[38] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: helping developers make informed logging decisions," in *Proceedings of 2015 IEEE/ACM IEEE International Conference on Software Engineering*, 2015, pp. 415–425.