2018 25th Australasian Software Engineering Conference (ASWEC)

How is Logging Practice Implemented in Open Source Software Projects? A Preliminary Exploration

Guoping Rong, Shenghui Gu†, He Zhang, Dong Shao, Wanggen Liu‡
State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University, Nanjing, China
{ronggp, hezhang, dongshao}@nju.edu.cn, †dz1732002@smail.nju.edu.cn, ‡wayne.liu@transwarp.io

***Abstract*—*Background:* Logs are the footprints that software systems produce during runtime, which can be used to understand the dynamic behavior of these software systems. To generate logs, logging practice is accepted by developers to place logging statements in the source code of software systems. Compared to the great number of studies on log analysis, the research on logging practice is relatively scarce, which raises a very critical question, i.e. as the original intention, can current logging practice support capturing the behavior of software systems effectively? *Aims:* To answer this question, we first need to understand how logging practices are implemented these software projects. *Method:* In this paper, we carried out an empirical study to explore the logging practice in open source software projects so as to establish a basic understanding on how logging practice is applied in real world software projects. The *density*, *log level (what to log?)* and *context (where to log?)* are measured for our study. *Results:* Based on the evidence we collected in 28 top open source projects, we find the logging practice is adopted highly inconsistently among different developers both across projects and even within one project in terms of the density and log levels of logging statements. However, the choice of what context the logging statements to place is consistent to a fair degree. *Conclusion:* Both the inconsistency in *density* and *log level* and the convergence of *context* have forced us to question whether it is a reliable means to understand the runtime behavior of software systems via analyzing the logs produced by the current logging practice.**

***Keywords*—log, logging practice, empirical study, Java-based**

I. Introduction

Logs are generally used to record the runtime behavior of software systems or services. A variety of software engineering tasks with diverse purposes depend on logs, for example, debugging, monitoring, auditing, defect prediction and so on [1]–[5]. In particular, logs are significant for software developers and testers to diagnose failures both in testing environment and production environment [6], [7]. Sometimes it is the only way for software engineers to deal with production failures. Moreover, with the rise of new technologies such as AIOps [8] in recent years, logs play an increasing important role to provide critical information for system operations.

To produce logs, *logging practice* is a software engineering practice that software developers used to put logging statements here and there in the source code they developed. The importance of logging practice has been widely recognized in industry [9]. Apparently, to be useful, logs generated by logging statements in the source code should be well-formed and informative, which requires logging practice to be carried out properly. However, it is normally difficult to make sound decisions to determine the context of logging statements (where to log?) and the content of logging statements (what to log?) [10]–[12]. For example, study [13] lists several scenarios, with which developers should not put logging statements. Meanwhile, the content of logging statement could also influence its capability to capture the runtime behavior of software systems. For example, study [14] reveals that more than half of logging statements even do not contain any variables. These facts may raise a critical issue, i.e. can current logging practice provide necessary and reliable support to produce logs for further analysis? To answer this question, we must first understand the implementation status of the logging practice in real software projects. To the best of our knowledge, there are very few studies in the academic community covering this topic. In this sense, we carry out this empirical study to explore the logging practice in real-world software projects. Since it is not likely to obtain the source code of commercial software systems, we mined 100 top open source projects on GitHub, from which we extracted logging statements to provide the data source for our investigation. By analyzing the logging statements, we identified obvious inconsistence both across projects and within each project in terms of *density* and *log level*. In addition, the placement of the logging statements converges. These facts to a fair degree imply that the logging practice has not been well carried out in these projects. In view of this, we have to question whether it is reliable and valid to capture program behavior by analyzing the logs.

The rest of the paper is organized as follows. Section II briefs the related work. Section III introduces the research objective and the approach we applied. Section IV reports the analysis results. Section V discusses the implications. The threats to validity are also presented in this section. Finally, we conclude this paper in Section VI with suggestions for future work.

II. Related Work

A. Log Analysis

In view of the importance of logs, research on this topic attracts more and more researchers. The majority of these researches concentrate on log analysis [15], which means

2377-5408/18/$31.00 ©2018 IEEE
DOI 10.1109/ASWEC.2018.00031

analysts utilize various techniques to analyze the existing logs and retrieve useful information to support decisions, e.g., causal paths [16], event correlations [7], [17], component dependency [18], resource usage [19], etc. Log analysis is also widely used in anomaly detection [1]–[3], [5], monitoring [4], cause diagnosis [6], [7], etc. In addition, tools supporting log collection and analysis spring up in industrial environment, such as *ELK* (Elasticsearch, Logstash, Kibana), *Splunk*, etc.

B. Researches on Logging Practice

Apparently, log with high quality is the prerequisite of useful log analysis, which requires logging practice being carried out properly. Nevertheless, there are quite few studies focusing on logging practice.

Yuan et al. conducted a series of studies on logging practice. For example, they confirmed the importance of logging practice with quantitative evidences [20]. In another study [16], they proposed an approach to enhance the existing logging statements to collect causally-related information so as to reduce the burden when diagnosing failures. In [21], Yuan et al. identified a number of patterns for logging practice to put logging statements so as to balance "over logging" and "insufficiently logging". To make logging practice more effective, they proposed to automate the placement of logging statements by measuring the degree of software uncertainty that can be removed by adding a logging statement [11]. The approach is able to compute an optimal logging statement placement, disambiguating the entire function call path with acceptable performance slowdown. Chen et al. [22] performed a replication study by Yuan et al. [20] and observed similar results in 21 Java projects. Furthermore, they characterized six anti-patterns in the logging statements by investigating the developing history of three open source systems [13]. Fu et al. studied logging practice of two large-scale online service systems at Microsoft [10]. They provided six findings on the categories of logged code snippets and factors for logging decisions. Based on the first step they made, they proposed a "learning to log" framework, aiming to learn the common logging practice automatically.

Apart from studies on log enhancement and logging decision, Kabinna et al. examined changes to logging statements in four open source applications in order to reduce the effort for maintenance [23].

In spite of the aforementioned effort made by researchers, logging has been still an arbitrary and subjective practice in industry. There are no well-defined and broadly-accepted logging guidelines for developers to refer to during software development [10], [24].

C. Tools Guiding Logging Practice

Up to now, tools supporting logging practice are not rare, e.g., *log4j*, *SLF4J*, etc. However, tools guiding logging practice are still relatively scarce in industry.

Tools to guide where to log: Zhu et al. implemented an automatic logging suggestion tool called *LogAdvisor*, which is able to recommend developers where to log and potentially reduce their logging effort. A similar work is conducted by Zhao et al. [25], they proposed an automated tool *Log20*, which determines a near optimal placement of logging statements within a predefined performance overhead. Chen et al. developed *LCAnalyzer*, which statically scans through the source code and searches for anti-pattern instances mentioned in [13] so as to avoid "bad smell" logging statements. Ding et al. presented *Log*^{2}, which is a cost-aware logging system to optimally decide "whether to log". This tool currently only works for performance monitoring and diagnosis.

Tool to guide what to log: Yuan et al. presented a tool named *LogEnhancer* [16] that modifies each logging statement in a certain piece of code to ease failure diagnosis by collecting additional causally-related information. In addition, they developed *Errlog* [21], a tool that inserts proactive logging statements to help developers capture more useful information for postmortem failure diagnosis.

III. Research Method

A. Research Objectives

Although there are multiple purposes for log analysis, we clearly define the scope of our study in the field of software engineering. To be specific, logs are used to understand the behavior of software systems so as to assist defect or issue detection. In this sense, those studies using logs to understand user habit/behavior so as to support recommendation are excluded in our study.

In recent years, technological innovations such as *cloud computing*, *microservices* and *service mesh*, etc. spring up, making the services and systems more than ever distributed and intricate, which greatly challenges the development and operation teams. By adopting logging practice, dynamic behaviors of software systems and services could be captured in logs, which provides valuable information for developers and maintainers to understand what happened during the runtime of software systems. Obviously, to satisfy this goal, all nontrivial information are supposed to be recorded in logs, which requires the logging practice to be carried out properly. Unfortunately, there are very few relevant studies in academia, which makes it impossible for us to understand the implementation of the logging practice in real-world software projects.

In light of this, we define the main research objective of our study is: *To explore the logging statements in real world software projects in a quantified manner so as to understand how logging practice is implemented in real-world software projects.*

To address this research objective, we defined three research questions.

First, the *density* of logging statements depicts the overall status of logging practice in projects. Although there is no wildly accepted criterion on suitable *density*, if it is too low, it is likely that the logging practice has not been well implemented. To this end, our first research question is:

- RQ1: what is the density and distribution of logging statements in these projects?

TABLE I: Description of Java log levels

Log Level	Description
Fatal	This level designates very severe error events that will presumably lead the application to abort.
Error / Severe	This level designates error events that might still allow the application to continue running.
Warn / Warning	This level designates potentially harmful situations.
Info / Config	This level designates informational messages that highlight the progress of the application at coarse-grained level.
Debug / Fine / Finer	This level designates fine-grained informational events that are most useful to debug an application.
Trace / Finest	This level designates finer-grained informational events than the "Debug".

Second, *log level* represents what to log. Usually, there are mainly six different log levels, as presented in Table I. Note that the levels are different among different tools. For example, in *log4j*, *log4j2* and *LOGBack*, "Fatal", "Error", "Warn", "Info", "Debug" and "Trace" are used to represent various levels, meanwhile, in *Java logger* and *SLF4J*, "Severe", "Warning", "Config", "Fine", "Finer" and "Finest" are used. In order to unifying the standard for log levels, we map the levels in these tools in light of the official document of SLF4J[1]. Basically, if the log level varies greatly among different projects or contributors within the same project, we may conclude that logging practice has not been consistently implemented. Therefore, we define the second research question as follows:

• RQ2: what is the distribution of log level in these projects?

Third, the context of logging statements addresses the locations where developers put logging statements, i.e. by which types of code snippet it is surrounded. We use the types defined in *JavaParser*, e.g., *If statement*, *For statement*, *Catch clause*, etc.[2] The basic idea is that if the distribution of logging context varies greatly among projects or contributors within the same project, it is more likely that that logging practice has not been consistently implemented. To this end, the third research question is:

• RQ3: what is the distribution of the context of logging statements?

B. Metrics and Analysis

To answer the three research questions, we defined several metrics.

1) The density of logging statements: This metric could be further divided into two sub-metrics: 1) D_a: the density of logging statements from the project perspective, 2) D_w: the density of logging statements from the contributor perspective.

As the term implies, *the density of logging statements* counts the number of logging statements per KLOC. Apparently, the density of logging statements in a certain project could be calculated via the following equation.

$$Density = \frac{\#Logging\ statement}{LOC\ of\ the\ source\ code}$$

[1]https://www.slf4j.org/apidocs/org/slf4j/bridge/SLF4JBridgeHandler.html

[2]please refer to http://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.6.5 for more details.

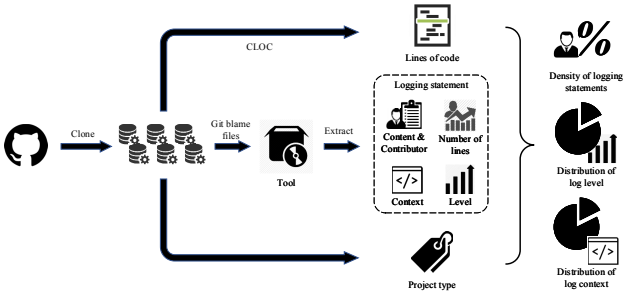

Fig. 1: Procedure of data preparation

If we group the logging statements and source code by distinct contributors, we can calculate the density of logging statements from the contributor perspective using the same equation similarly.

2) The log level of logging statements: This metric consists of two sub-metrics: 1) L_a: the log level of logging statements from the project perspective, 2) L_w: the log level of logging statements from the contributor perspective.

The metric of *the level of logging statements* is then calculated as the percentage of the number of logging statements with a certain logging level (e.g., Fatal, Error, etc.) to the total number of logging statements.

3) The context of logging statements: This metric could also be further elaborated in two sub-metrics: 1) C_a: the context of logging statements from the project perspective, 2) C_w: the context of logging statements from the contributor perspective.

Therefore, the metric of *the context of logging statements* is calculated as the percentage of logging statements with a certain context type both in terms of project and contributor.

C. Data Preparation

We focus our investigation on open source projects, since the source code of commercial software systems is inaccessible. The whole procedure of data preparation is shown in Figure 1. In a nutshell, we first retrieve the source code of each project from GitHub. Then we extract the necessary data (i.e., logging statements) from the source code. Finally, we apply descriptive statistics to analyze the data and form findings.

1) Data Source: We use one of the largest online repositories (i.e., GitHub) as the source from which we retrieved data for investigation. GitHub utilizes tool Git for version control and source code management, which provides plenty of commands for us to retrieve related information for our investigation. Moreover, GitHub provides abundant resource for us to investigate the status that contributors carrying out logging practice. For example, a significant feature that Git provides is the `git blame`, which allows us to differentiate the owner of each line of code. Open source projects on GitHub are ranked by stars, which represents the number of people who like the project. To some extent, the number of stars one project received reflects the degree of population.

TABLE II: Projects on GitHub after screening

ID	Project	Rank	Type	Description
P1	ReactiveX/RxJava	2	Library	RxJava - Reactive Extensions for the JVM - a library for composing asynchronous and event-based programs using observable sequences for the Java VM.
P2	elastic/elasticsearch	3	Product	Open Source, Distributed, RESTful Search Engine
P3	spring-projects/spring-boot	6	Framework	Spring Boot
P4	google/guava	7	Library	Google core libraries for Java
P5	spring-projects/spring-framework	12	Framework	Spring Framework
P6	apache/incubator-dubbo	15	Framework	Apache Dubbo (incubating) is a high-performance, java based, open source RPC framework.
P7	skylot/jadx	23	Decompiler	Dex to Java decompiler
P8	libgdx/libgdx	25	Framework	Desktop/Android/HTML5/iOS Java game development framework
P9	netty/netty	26	Framework	Netty project - an event-driven asynchronous network application framework
P10	Netflix/Hystrix	27	Library	Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.
P11	alibaba/fastjson	28	Library	A fast JSON parser/generator for Java
P12	alibaba/druid	36	Library	Druid is one of the best database connection pools written in Java. Druid provides powerful monitoring functionalities and more.
P13	SeleniumHQ/selenium	39	Framework	A browser automation framework and ecosystem.
P14	shuzheng/zheng	44	Framework	A distributed system architecture for agile development based on Spring+SpringMVC+Mybatis.
P15	nathanmarz/storm	59	Framework	Distributed and fault-tolerant realtime computation: stream processing, continuous computation, distributed RPC, and more
P16	bazelbuild/bazel	62	Product	a fast, scalable, multi-language and extensible build system
P17	EnterpriseQualityCoding/FizzBuzzEnterpriseEdition	63	Game	FizzBuzz Enterprise Edition is a no-nonsense implementation of FizzBuzz made by serious businessmen for serious business purposes.
P18	deeplearning4j/deeplearning4j	64	Library	Deep Learning for Java, Scala & Clojure on Hadoop & Spark With GPUs - From Skymind
P19	openzipkin/zipkin	68	Product	Zipkin is a distributed tracing system
P20	apache/kafka	69	Product	Mirror of Apache Kafka
P21	eclipse/vert.x	73	Framework	Vert.x is a tool-kit for building reactive applications on the JVM
P22	LMAX-Exchange/disruptor	78	Library	High Performance Inter-Thread Messaging Library
P23	monkeyWie/proxyee-down	81	Product	Http download tool based on Http proxy
P24	prestodb/presto	82	Product	Distributed SQL query engine for big data
P25	mybatis/mybatis-3	85	Framework	MyBatis SQL mapper framework for Java
P26	perwendel/spark	87	Product	A simple expressive web framework for java. News: Spark now has a kotlin DSL
P27	clojure/clojure	97	Language	The Clojure programming language
P28	apache/hadoop	100	Product	Mirror of Apache Hadoop

Since Java is one of the most popular programming languages for back-end development, we select top 100 Java projects based on stars for our investigation. Through the following URL, we can obtain the list of projects sorted by stars, and the top 100 projects (up to May, 2018) are the target source of our investigation in the end.

```
https://github.com/search?l=Java&o=desc&q=stars%3A%3E1&s=
stars&type=Repositories&utf8=%E2%9C%93
```

To be consistent and easy to compare, we only consider native Java language. Therefore, projects using *Groovy*, *Scala*, and *Clojure* are excluded. Moreover, android projects are also excluded in our research, since developers may minimize the use of log statements to fit the application environment, i.e., mobile phones. Besides, projects without source code and for tutorial purpose are also excluded. As the result, we eventually retain 28 projects. Table II lists the 28 projects with relevant information. We downloaded all projects into our local environment for further processing via shell command `git clone`. And in this paper, we use character "*P*" followed by numbers to refer to each project in the following sections.

2) Data Extraction: We aim to retrieve the following information from each project.

- content of each logging statement
- corresponding contributor of each logging statement
- log level of each log statement
- context of each log statement
- lines of code (LOC) of each project
- type of each project

To meet this purpose, we developed a tool, Java Log Retriever (JLR)[3] to extract these information. Generally, the data extraction procedure consists of three main aspects, i.e. 1) Generating intermediate information; 2) Data retrieval from source code; 3) Meta-data retrieval from projects, as shown in Figure 1. In the following paragraphs, we will illustrate each step in detail.

a) ***Step 1: Generating intermediate information****:* Our first step is to generate a map from contributors to code, as one of our research intentions is to understand the different logging preference between distinct contributors. Based on this map, each line of code can be attributed to its corresponding contributor. This step is the cornerstone of further extraction and analysis, providing us an approach to know how many lines of logging statements a certain contributor wrote and what kind of logging statements he or she put in code, and so on. This step utilizes several shell and Git commands to finish the operation, which are sequenced by Linux pipeline. The command for Git blame file generation is as follows:

[3]We had it open sourced on Github web through https://tinyurl.com/ybucf6g9

```
git ls-files | grep -E "\.java$" | sed 's/\(.*\.\)java$/git
    ↪ blame \1java > \1blame/' | sh
```

Basically, the command consists of four steps. In the first place, we utilize Git command - `git ls-files` to list all files in Git index and working tree. And then Java files are preserved while other types of files are omitted, since we only investigate Java logging statements. In the next part, we use `git blame` to get the information of each line, e.g., author, time, path. `sed` command is used to transform Java file name into `git blame` command which outputs the final files with ".blame" extension. In the end, the previous command generated by `sed` command is executed via `sh` command.

We run this command in the root path of each project and each Git blame file is generated under the same path of its corresponding Java file. Finally, the example results after retrieving are like the following.

```
043881148 multiton/src/main/java/com/iluwatar/multiton/App.
java (daniel-bryla 2016-10-23 19:59:03 +0200 50) LOGGER.info
("MURAZOR={}", Nazgul.getInstance(NazgulName.MURAZOR));
```

The first part of this line is 40-byte SHA-1 of the commit the line is attributed to. The next part is the path of the file the logging statement belongs to. In the parentheses, the author name, timestamps and the line number are given. The last part is original statement.

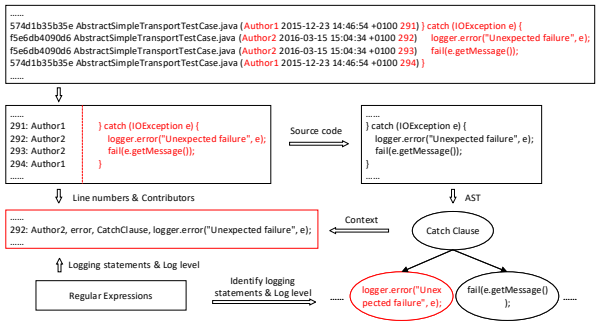

Fig. 2: Process of data retrival

b) ***Step2: Data retrieval from source code:*** It is not likely that the context of logging statements could be extracted directly from source code. Prior to retrieve the context of logging statements, we need to know the code structure, e.g., in which code snippet each logging statement is located. To achieve this goal, we utilize Abstract Syntax Tree (AST) to represent the code structure of Java file (i.e. file with ".java" extension).

The aforementioned process is the principle of how JLR works in general. For more detail, as shown in Figure 2, JLR first reads the Git blame files generated in previous procedure. Next, it records the line number, corresponding author and content of each statement. And then, given all statements, it parses them into an AST with the aid of JavaParser[4]. The next step is to identify whether the expression statement is a logging statement or not, and we use the following regular expressions to identify logging statements.

```
(?i)log[^. ]*\.[^. ]*(fatal|error|warn|info|debug|trace)[^.
    ↪ ]*
(?i)log[^. ]*\.log\((severe|warn|info|config|fine|finer|
    ↪ finest)
(?i)log[^. ]*\.log\(level\.(severe|warn|info|config|fine|
    ↪ finer|finest)
```

If a logging statement node is identified, JLR first retrieves its parent node and then get the type of the parent node. Meanwhile, log level of logging statement is extracted and recorded, as well as its content.

Finally, JLR outputs the data generated from each of Git blame files and merges them into one final report. The format of the report is described below.

```
426: cpovirk, SEVERE, CatchClause, log.log(Level.SEVERE, "
RuntimeException while executing runnable " + runnable + "
with executor " + executor, e);
```

The part before the colon is the line number, followed by its contributor, level and context. The rest part is the original logging statement.

c) ***Step3: Meta-data retrieval from project:*** In this step, we need to retrieve meta-data information from each project, i.e. lines of Java code and the type or category of project. As for retrieving lines of Java code, we count the lines of code in each project, utilizing an open source tool named *CLOC*[5]. The purpose of logging may vary among different types of projects. In this sense, we manually identify the types of projects according to the description on their homepage. A noteworthy point is that the project types only provide a rough dimension for understanding logging practices. In other words, the same type of projects in our study does not necessarily adopt the same logging strategy.

IV. Results

Based on the data we extracted from the 28 projects, we perform descriptive statistic to show the status of logging practice in these projects.

A. Status and Difference Across Projects

We find the logging behavior and preferences of contributors in various projects are different in terms of *density* and *log level*. However, the *context* of logging statements converges.

1) D_a: Density and Distribution of Logging Statements Across Projects: Among the projects listed in II, most of the projects have logging statements except three projects, i.e. *P1*, *P17* and *P27*. We removed these three projects for the finial analysis.

In general, we defined two major types of projects, i.e. *libraries & frameworks* and *products*, respectively. The former type of projects are primarily used at development stage to be called by other software systems. On the contrary, the latter type of projects could be used as a standalone systems. Considering that different projects may have distinct purposes for logging, we perform statistic analysis these two types of projects separately. Figure 3 and Figure 4 depict the details of

[4]http://javaparser.org/

[5]https://github.com/AlDanial/cloc

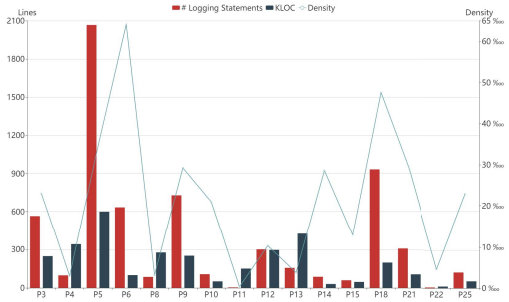

Fig. 3: Density of logging statements in libraries & frameworks

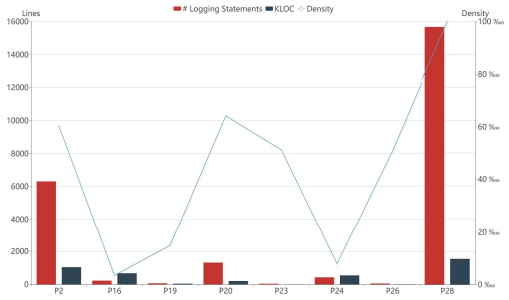

Fig. 4: Density of logging statements in products

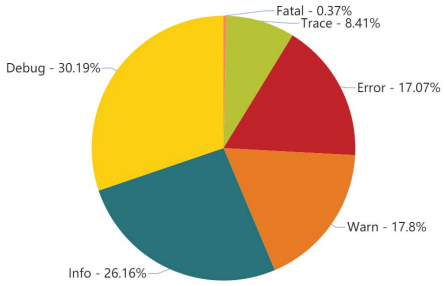

Fig. 5: Distribution of log level in libraries & frameworks

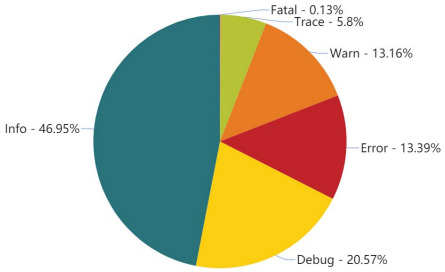

Fig. 6: Distribution of log level in products

the different two categories , including the number of logging statements, lines of code and density of logging statements.

The broken line represents the density of logging statements of distinct projects. And two bars with different colors represent the number of logging statements (red bar) and lines of code (indigo bar) respectively. We should notice that the unit of lines of code in Figure 3 and Figure 4 is KLOC, and this metric includes only physical code without comments.

Apparently, in both types of projects using logging technique, the density of logging statements turns out to be quite different. The density of logging statements in some open source projects is at a very low level, e.g., *P4*, *P8*, *P11*, *P13* and *P16*. Besides, the density of logging statements in most open source projects is distinctly lower than that in commercial software projects [12] (roughly one logging statement per 58 LOC), indicating a relatively insufficient logging in open source projects. We believe the different culture between open source projects and commercial projects may have led to this phenomenon. In commercial software projects, the use of logging statements is often regulated through relatively stricter coding conventions and code reviews. In contrast, in open source projects, the implementation of similar conventions is relatively relaxed.

Moreover, it is obvious that the density of logging statements varies among different projects, ranging from 0.2‰ to 100‰. To be more intuitive, in some projects, there is only one logging statement in every 50,000 lines of code while in other projects there is one logging statements in every 100 lines of code. This tremendous difference may imply inconformity among the contributors in different projects when applying logging practice.

Finding 1: The density of logging statements is generally low and varies tremendously across different projects.

2) L_a: Distribution of Log Level: Given the logging statements extracted from each project, we analyze each logging statement to identify what the log level is. We depict the distribution of log level in *libraries & frameworks* as well as in *products*, as shown in Figure 5 and Figure 6 respectively. The log levels include “Fatal”, “Error”, “Warn”, “Info”, “Debug”, “Trace”, and we map “Severe”, “Warning”, “Config”, “Fine”, “Finer” and “Finest” into its corresponding level, according to Table I.

Overall, the distribution of different log levels varies greatly among the two types of open source projects. In the type of *libraries & frameworks* projects, “Debug” and “Info” are the most two log levels, occupying 30.19% and 26.16%, respectively. In the type of *products* project, the first two are still “Debug” and “Info”, but the positions are reversed, i.e., log level of “Info”, accounting for 49.95% and “Debug” 20.57%. Besides, in the two types of projects, four types of log level (i.e. “Debug”, “Info”, “Warn”, and “Error”) account for over 90% of the total log levels, indicating these four types of log levels are most preferred by open source developers. It should be noted that this does not mean that the above four log levels generate the most logs during runtime. For example, a manual checking reveals that a lot of logging statements on “Debug” level are embraced by “if” statement, which could be turn off in the production environment.

To further explore the log levels, we pick out the projects

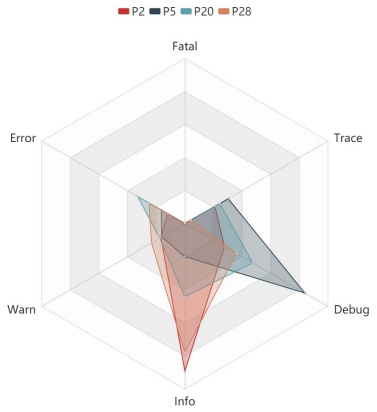

Fig. 7: Difference on log level

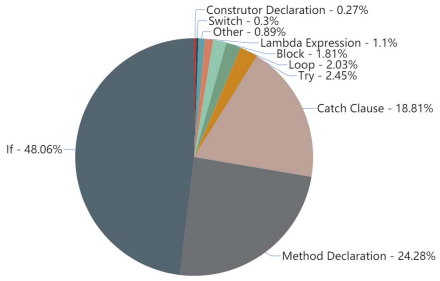

Fig. 8: Distribution of log context in libraries & frameworks

which have over one thousand lines of logging statements and depict the distribution of log level in one radar chart, as shown in Figure 7. Note that *P2*, *P20* and *P28* share the same project type. We could also observe major differences with respect to the log levels applied in these projects.

Finding 2: The distribution of major log levels are different among various projects.

3) C_a: Distribution of Log Context: The log context in two types of projects are shown in Figure 8 and Figure 9. To simplify, we merge several similar context description into one. Specifically, in Figure 8 and Figure 9, "Switch" is a combination of "Switch Statement" and "Switch Entry Statement"; "Loop" is a combination of "Foreach Statement",

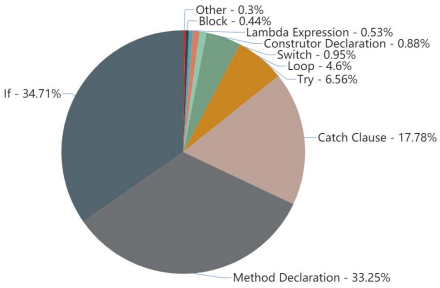

Fig. 9: Distribution of log context in products

"For Statement", "While Statement" and "Do Statement". It is obvious that developers prefer to add a logging statement in a "If Statement" in both types of projects. "Catch Clause" usually implies that certain exceptions have occurred during runtime, which attracts developers to put log statements to understand what happened. Meanwhile, it may be a performance consideration that there are relatively few (less than 5%) log statements in "Loop" structure.

Finding 3: Developers tend to add log statements to code snippets that have branches, e.g., "If Statement" and "Try-Catch" clause.

B. Status and Difference Within One Project

Preferences to carry out logging practice of various contributors seem to be different across different projects. Since this phenomenon may be due to differences among projects, we also investigated the implementation of logging practices in the same project.

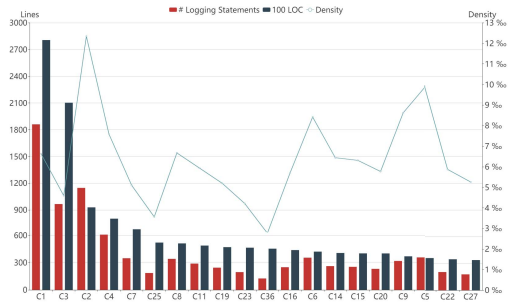

Fig. 10: Logging statements from different contributors within the same project

1) D_w: Density of Logging Statements: Even within the same project, logging preferences of distinct contributors are different as well. We choose *P28* as an example [6]. The results are presented in Figure 10. We treat contributors anonymously to avoid revealing privacy by denoting them with character "*C*" followed with numbers, which denotes the rank according to the number of logging statements in their source code. For example, *C1* wrote the most log statements in his code. Besides, contributors are sorted by the size (measured by LOC) of the source code they contributed to this project in descending order. As Figure 10 shows, we listed top 20 contributors in this figure. The red bar represents the number of logging statements while the indigo bar represents the lines of code they contributed.

It can be observed that different contributors do not have the similar behavior to conduct logging practice. Some of them log more frequently, for instance, as the third contributor, *C3* put nearly one logging statement in every 81 lines of code. Meanwhile, the second contributor put one logging statement nearly in every 218 lines of code. In general, we could observe relatively large difference with respect to the

[6]please refer to https://tinyurl.com/y8hjh7sn for complete results.

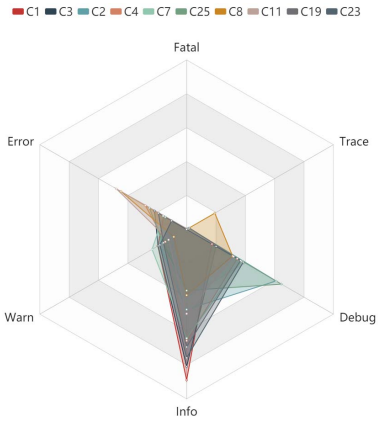

Fig. 11: Distribution of log level in P28

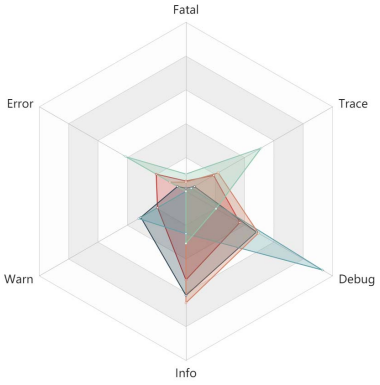

Fig. 12: Distribution of log level in P3

density of logging statements for the top 20 contributors in *P28*, ranging from 2.77‰ to 12.35‰.

Finding 4: The density of logging statements from distinct contributors is different among various contributors within the same project.

2) L_w: Distribution of Log Level: Take *P28* as an example, we analyze the distribution of the log level from the top 10 contributors. As shown in Figure 11, the log levels they applied are different from each other. For example, *C11* prefer to use "Trace" more than others. Similar phenomenon could be observed in *P3* (Figure 12).

In general, major differences can be observed among the top 10 contributors in terms of the log level the applied in their source code. Similar phenomenon can also be found in other projects [7].

Finding 5: Even within a project, the log level of distinct contributors is different in most cases.

3) C_w: Distribution of Log Context: The distribution of log context between distinct contributors seems to be similar, as shown in Figure 13. Obviously, contributors prone to place

[7]see https://tinyurl.com/y8hjh7sn

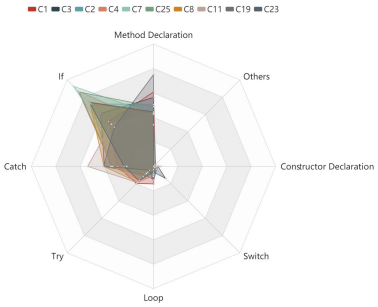

Fig. 13: Distribution of log context in P28

logging statements inside "If" blocks, more generally, inside potentially new branches.

Finding 6: Within a project, the distribution of log context seems to be similar among different contributors.

V. Discussions

By extracting and analyzing the logging statements in most popular projects, we identified major difference on the *density* and *log level* of logging statements, as well as the convergence of logging *context*, which to a fair degree imply that logging practice has not been well carried out in these open source projects.

A. Implications

Given the aforementioned results, there are several important implications.

Ad-hoc logging: It seems that the logging practice among distinct developers differ from each other to a very large extent. Not only the density of logging statements, but also the contents (what to log?) of logging statements vary among different developers. With this ad-hoc manner to carry out the logging practice, we have to doubt whether logs can provide a reliable source of information for log analysis to establish a correct understanding of the runtime behavior of the program.

Balance and trade-off of logging practice: Given the current status on logging practice in open source projects, it might be too early to discussion the "balance" and "trade-off" when conducting logging practice. Although this is a very hot topic and big challenge in research towards better logging practice [10], [12], [26], [27], the huge difference on logging *density* and *levels*,together with the very few *contexts* suggest that before everything, suitable mechanisms to ensure the logging practice being well conducted should be devised.

Research on Log analysis: Similarly, as another hot research topic, log analysis attracts many attention among software engineering practitioners [3], [15], [19]. Nevertheless, as revealed in this study, current logging practice in open source projects might not provide quality logs to support log analysis well. Due to insufficient or even misleading logs generated by current logging practice, it is not likely to mine valuable information from them. In this sense, we suggest that relevant researchers should pay more attention to logging

practice and improve the quality of logs before exploring more sophisticated approaches to perform log analysis.

B. Possible Reasons

We discuss several possible reasons for the current status on logging practice in open source projects in this subsection.

Lacking awareness to perform logging practice: It is not rare that some contributors did very little (even none) logging practice in these projects. Note that even if the log mechanism is turned off in the production environment, our tools can still extract the corresponding log statement, unless the logging statements have been removed or not written at all. In this sense, lacking the awareness to perform logging practice seems to be one of the reasons for this phenomenon. Besides, as studies [13], [23] pointed out, along with system evolution, logging practice is easily being neglected, which also requires strengthened awareness to perform logging practice.

Lacking practical guidelines: It seems that there lacks of practical guidelines for software developers to perform logging practice. As the result, personal experience and preference plays an important role in logging practice in these projects. This is reflected in the difference in the density, levels and context of logging statements.

Lacking suitable tools: Another reason for this phenomenon might be lacking of tools to guide logging practice and check logging statements afterwards. Without suitable tools, software developers have to rely on manual checking according to personal experience, which is not only time-consuming but also error-prone.

C. Next Step

It seems that practical and context-specific guidelines for logging practice is in urgent need to help developers conducting this practice. There are several existing guidelines such as [28], [29]. However, most of these guidelines work for general purpose, which may not be suitable for a specific project. Besides, how to implement these guidelines is also a big challenge, which lead to another much-needed work, i.e., supporting tools to guide the logging practice. Several studies proposed tools to help developers put logging statements (cf. subsection II-B), however, tools to perform checking for logging statements according to predefined guidelines are also needed to implement the logging practice properly.

D. Threats to Validity

a) Java-based projects: Our study is performed on Java-based open source projects and we did not investigate the projects based on other languages. Which may to a certain degree impact our conclusion. Nevertheless, Java is one of the most adopted programming languages in the open source community. In this sense, the result in this study could reflect the status of logging practice in open source projects to a fair degree.

b) Various application areas of projects: The projects we investigated are all from GitHub, including some famous projects from big organizations and some from individuals. From the results of the study, the obvious differences in log density and levels and the convergence of the context of logging statements are a common phenomenon. In this sense, our findings and conclusions, to a fair degree, still valid.

c) Data extraction: The procedure we used to extract the source code may not be able to capture all logging statements. At the current stage, JLR only supports logging statements generated by *Log4j*, *Log4j2*, *SLF4J*, *Java logger* and *LOG-Back*. Therefore, there may be some logging statements we omitted more or less. However, through manually checking, all the logging statements in the pilot extraction have been detected by JLR, which to a fair degree mitigate the impacts derived by missing logging statements.

d) Metrics to measure the status of logging practice: Our work implied that the logs generated by the existing logging practice may not be enough for us to reliably understand the program behavior. However, there is a conceptual gap here, i.e., the metrics we use can only reveal the differences in *density* and *level* among projects and contributors as well. However, this does not mean that the capture of program behavior by the logs generated by current logging practice must be inadequate and inaccurate. Nevertheless, given the nontrivial difference we identified on logging practices as well as the very few types of places (i.e., the *context* of logging statements), it is hard to establish the confidence that the runtime behavior could be properly captured.

VI. Conclusion

With the ever-increasing scale and complexity of software systems and services, to understand how these software systems and services operate is more and more important for developers and maintainers. As the footprint of running software systems and services, logs are one of the most important sources for software developers and maintainers to understand the runtime behaviors and identify issues. To satisfy this purpose, logs are supposed to contain sufficient information regarding the runtime behaviors, which requires the logging practice to be properly conducted. However, very few studies have been conducted to understand how well the logging practice has been carried out. In this sense, we carried out an empirical study to explore the logging practice in top open source projects on GitHub. The contribution of this study may be highlighted as follows:

First, instead of taking for granted that logs provides enough information to understand runtime behavior, we focus on the quality of logging practice which produces the logs, which seems to be neglected by the academic community currently. We noticed a blog post and the investigation behind [14], [30] also questioned the quality of the log, yet only very little information provided by this work which limits our capability to understand how logging practice has been implemented in real-world projects.

Second, through our empirical investigation, we revealed major difference in terms of *density* and *log levels* of logging statements as well as the convergence of logging statements over very few *contexts* both across projects and within one project, which implies that this practice has not been well conducted, leading to questionable capability to capture the

runtime behavior of software systems and services. A more profound implication for this fact is that it may impair the trustworthiness of the results came from current log analysis.

Last but not least, based on the results and related discussion, we point out that the main problem at present is to devise mechanisms to guarantee the quality of logs through properly logging practice. For example, to establish suitable guidelines and develop workable checking tools to help developers to do better logging practice.

There are still several limitations for this study at this preliminary stage, therefore, we suggest two major topics for future work. 1) To extend investigation scope to include more software projects (perhaps including commercial projects) so as to portray the adoption status of logging practice more comprehensively. 2) We did not conduct survey to explore how developers conduct logging practice in this study. Therefore, one promising future work might be a survey to better understand how the logging practice is conducted and how to improve this practice.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (Grant No.61572251).

REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. New York, NY, USA: ACM, 19 October 2003, pp. 74–89.

[2] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. New York, NY, USA: ACM, 23 October 2005, pp. 105–118.

[3] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 Ninth IEEE International Conference on Data Mining (ICDM '09)*. IEEE, 6 December 2009, pp. 149–158.

[4] M. Montanari, J. H. Huh, D. Dagit, R. B. Bobba, and R. H. Campbell, "Evidence of log integrity in policy-based security monitoring," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN '12)*. IEEE, 25 June 2012, pp. 1–6.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. New York, NY, USA: ACM, 11 October 2009, pp. 117–132.

[6] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *2008 19th International Symposium on Software Reliability Engineering (ISSRE '08)*. IEEE, 10 November 2008, pp. 117–126.

[7] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. Berkeley, CA, USA: USENIX Association, 25 April 2012, pp. 26–26.

[8] J. Hertvik and S. Paskin. (2017) What is AIOps? AIOps explained. [Online]. Available: https://www.bmc.com/blogs/what-is-aiops/

[9] B. W. Kernighan and R. Pike, *The practice of programming*. Addison-Wesley Professional, 1999.

[10] 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 the 36th International Conference on Software Engineering (ICSE Companion 2014)*. New York, NY, USA: ACM, 7 June 2014, pp. 24–33.

[11] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "The game of twenty questions: Do you know where to log?" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. New York, NY, USA: ACM, 2017, pp. 125–131.

[12] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15)*. IEEE, 16 May 2015, pp. 415–425.

[13] B. Chen and Z. M. J. Jiang, "Characterizing and detecting Anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 71–81.

[14] H. Idan. (2017) GitHub research: Over 50% of java logging statements are written wrong. [Online]. Available: https://blog.takipi.com/github-research-over-50-of-java-logging-statements-are-written-wrong/

[15] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Communications of the ACM*, vol. 55, no. 2, pp. 55–61, February 2012.

[16] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 4:1–4:28, 1 February 2012.

[17] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, and G. Lu, "LogMaster: Mining event correlations in logs of Large-Scale cluster systems," in *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, 8 October 2012, pp. 71–80.

[18] A. J. Oliner and A. Aiken, "Online detection of multi-component interactions in production systems," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. IEEE, 18 July 2011, pp. 49–60.

[19] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in Google compute clusters," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. New York, NY, USA: ACM, 26 October 2011, pp. 3:1–3:14.

[20] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in Open-Source software," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. Piscataway, NJ, USA: IEEE Press, 2 June 2012, pp. 102–112.

[21] D. Yuan, S. Park, P. Huang, Y. Liu, M. M.-J. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 8 October 2012, pp. 293–306.

[22] B. Chen and Z. M. J. 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. 330–374, 1 February 2017.

[23] S. Kabinna, W. Shang, C. P. Bezemer, and A. E. Hassan, "Examining the stability of logging statements," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, March 2016, pp. 326–337.

[24] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15')*. IEEE, 16 May 2015, pp. 169–178.

[25] 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 the 26th Symposium on Operating Systems Principles (SOSP '17)*. New York, NY, USA: ACM, 2017, pp. 565–581.

[26] F. Baccanico, G. Carrozza, M. Cinque, D. Cotroneo, A. Pecchia, and A. Savignano, "Event logging in an industrial development process: Practices and reengineering challenges," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW '14)*. IEEE, 3 November 2014, pp. 10–13.

[27] 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, 14 October 2016.

[28] Stephan. (2008) 7 good rules to log exceptions. [Online]. Available: http://codemonkeyism.com/7-good-rules-to-log-exceptions/

[29] C. Eberhardt. (2014) The art of logging. [Online]. Available: https://www.codeproject.com/Articles/42354/The-Art

[30] A. Zhitnitsky. (2016) 779,236 Java logging statements, 1,313 GitHub repositories: ERROR, WARN or FATAL? [Online]. Available: https://blog.takipi.com/779236-java-logging-statements-1313-github-repositories-error-warn-or-fatal/