Understanding Log Lines Using Development Knowledge

Weiyi Shang†, Meiyappan Nagappan†, Ahmed E. Hassan†, Zhen Ming Jiang‡
School of Computing, Queen's University, Canada†
Department of Electrical Engineering & Computer Science, York University, Canada‡
Email: {swy, mei, ahmed}@cs.queensu.ca†, zmjiang@cse.yorku.ca‡

***Abstract*—Logs are generated by output statements that developers insert into the code. By recording the system behaviour during runtime, logs play an important role in the maintenance of large software systems. The rich nature of logs has introduced a new market of log management applications (e.g., Splunk, XpoLog and logstash) that assist in storing, querying and analyzing logs. Moreover, recent research has demonstrated the importance of logs in operating, understanding and improving software systems. Thus log maintenance is an important task for the developers. However, all too often practitioners (i.e., operators and administrators) are left without any support to help them unravel the meaning and impact of specific log lines. By spending over 100 human hours and manually examining all the email threads in the mailing list for three open source systems (*Hadoop*, *Cassandra* and *Zookeeper*) and performing web search on sampled logging statements, we found 15 email inquiries and 73 inquiries from web search about different log lines. We identified that five types of development knowledge that are often sought from the logs by practitioners: meaning, cause, context, impact and solution. Due to the frequency and nature of log lines about which real customers inquire, documenting all the log lines or identifying which ones to document is not efficient. Hence in this paper we propose an *on-demand* approach, which associates the development knowledge present in various development repositories (e.g., code commits and issues reports) with the log lines. Our case studies show that the derived development knowledge can be used to resolve real-life inquiries about logs.**

I. Introduction

Logs play an important role in the maintenance of large software systems [1]–[3]. Logs report the major system activities (e.g., system workload or errors) and their associated contexts (e.g., a time-stamp or a user ID) to assist practitioners (i.e., operators and system administrators) in understanding the high-level system behavior in the field. Logs capture developers' expertise, since logging statements are inserted into specific code spots that are considered to be particularly important by developers or of great interest by practitioners [4]. Thus logging statements are an integral and necessary part of the code and maintaining them is just as important as maintaining the functional part of the code [5].

The rich yet unstructured nature of logs has introduced a new market of log management applications (e.g., Splunk [6], XpoLog [7] and logstash [8]) that assist in storing, querying and analyzing logs. Moreover, recent research has demonstrated the importance of logs in understanding and improving the quality of large scale software systems. For example, practitioners can leverage the rich information in the logs to generate workload information for capacity planning of large-scale systems [9], to monitor system health [6], to detect abnormal system behaviours [1], [10], [11], or to flag performance degradations [2], [12].

However, all too often practitioners are faced with many challenges in trying to understand the meaning of logs or to answer questions about specific log lines [13]–[15]. Unfortunately, there exists no research in understanding logs, even though they are of great importance. For example, there is an email inquiry about the log line "Too many fetch-failure" from *Hadoop* in its user mailing list[1]. In the email inquiry, the practitioner posted the log lines and asked "*... I wonder what's the causes of so many these error reports. Do you have similar experience? Or do you have any suggestions about my problem?*". The email reply to the inquiry did not answer the question, but suggested practitioner to search his answers on the web (i.e., *Google*).

We found that documenting all log lines is an inefficient use of the developers' time (since less than 1% of the log lines are inquired about). Additionally, knowing which lines to add more documentation is a difficult challenge, since we found log lines with all logging levels are inquired in practice and different users want to inquire about different log lines. Hence, we cannot simply add documentation to a specific set of log lines like error level logs. However, we still need to be able to help the practitioners with their log inquiries. Hence in this paper, we propose a systematic *on-demand* approach that attaches development knowledge to logs. Interpreting the attached development knowledge assists developers in understanding log lines. We define development knowledge of log lines as the information that is not conveyed directly in the log lines, but hidden in the development history of the code surrounding the logging statements from which these log lines are generated. Various sources of data generated during software development, such as development history [16], [17], design rationale, concerns of the source code [18], [19] and email discussions [20], [21] are widely used in program comprehension tasks.

The contributions of this paper are:

- This is the first work that proposes a systematic *on-demand* approach to help practitioners understand log lines. Our approach assists in resolving real-life log inquiries and outperforms the web search results.
- This is the first work that provides taxonomy on user inquiries of logs. We have identified five types of information that are often sought about log lines by practitioners - *meaning, cause, context, impact*

[1]http://goo.gl/jzEURG, checked April 2014.

TABLE I: Overview of the subject systems

System	Application domains	KLOC	Length of history	# logging statements
Hadoop	Distributed platform	580	8 years	5,641
Cassandra	Distributed database	118	4 years	1,080
Zookeeper	Distributed coordination service	78	5 years	1,163

and solution. We have derived this information by manually going through the user mailing list emails and StackOverflow questions for *Hadoop*, *Cassandra* and *Zookeeper*.

- Our derived development knowledge can also help identify experts regarding a particular log line.

The rest of this paper is organized as follows: Section II explains our process and the results of identifying the common types of information sought about log lines by the practitioners. Section III presents our approach that automatically extracts and synthesizes development knowledge for log lines on-demand. Sections IV and V evaluate our approach against three open source systems. Section VI discusses the related work. Section VII discusses the threats to validity. Finally, Section VIII concludes the paper.

II. Categorizing Log Inquiries

Understanding log lines is critical for practitioners. However, before we propose an approach to help practitioners understand log lines, we first need to better understand the types of information that are often sought about logs lines by the practitioners. Hence, we first perform an exploratory study that examines several real-life inquiries about log lines.

A. Manually Examining the Mailing Lists

We choose three subject software systems, which generate large amounts of logs during execution. Table I shows an overview of our subject systems. The systems are of different sizes, histories and application domains. However, they are all "systems software" (i.e., no user interfaces) – a choice that was done to ensure that these systems make heavy use of logging. We manually read through all the email threads in the mailing lists of the subject systems – a process that took over 100 man-hours. We end up with 15 different inquiries for log lines. For example, in the *Cassandra* user mailing list, an inquiry asks the following question:

Is it affecting my data put? (I have seen other weird validation exceptions where my data is still put and I can read it from cassandra and I get no exception client side)

Through our manual inspection, we have learned the following about asking and answering of log-related questions on the mailing lists:

1) Inquiries about logs may not be answered or might take a long period of time to be resolved. Although 12 of the 15 email inquiries had replies, practitioners never got an answer for 3 log lines in the mailing list. The maximum time for the first reply is over 105 hours.

2) Not all email replies are helpful. Some email replies are informative (e.g., resolving the issue), some replies are brief, some replies lack certainty and some replies are not useful. For example, in a reply to an inquiry, a user indicated that his/her prior reply was wrong and asked others to ignore the reply.

3) Manual analysis of the development knowledge is used to answer questions in some cases. For example, to find out the cause of a *Hadoop* log line, the practitioner manually browsed the source code of *Hadoop* and found the method that generates the log line. He was not familiar with Java, but after he posted the code snippets online, an expert replied and resolved his issue.

B. Identifying Types of Inquired Information

Based on these 15 threads, we have identified five different types of inquired information that are often sought about a log line by following the coding approach widely used by previous studies [22]. We repeat the process until we cannot find any new types of inquired information. We assign the "N/A" tag to an inquiry if the practitioners only include the log lines in the email without asking any type of particular information. Our study finds that there exist five types of information that are often sought about log lines. Table II tabulates the meaning of these five different types of inquired information.

We do note that a few industry guideline documents for creating logs do mention some of the types of information that we discovered above. For example, the design documents from the SANS consensus project for information systems [23] suggests that log lines should include the subject and object of the event, time of the event, tools that performed the event and the error status of the event. However, other types of information, such as the solution and impact, are not suggested. Moreover, developers may forget to provide some information even though it is required [24].

C. Cross-validating Our Findings

We were a bit surprised with the small number of inquiries about log lines on the mailing lists of the studied open source projects, since in prior industrial collaborations we noted that log lines played a key role in interactions between customers and developers (e.g., [25]). Hence, we randomly sampled 300 logging statements from the three subject systems and searched for the text in the logging statements using *Google*. If the text in the logging statement is ambiguous, we added the name of the subject system after the logging statement. For example, the text "Child Error" in a *Hadoop* logging statement may be ambiguous. We then search for "Child Error Hadoop" instead. We then examined the first 10 web search results from *Google*.

By examining the web search results, we found that 32, 23 and 18 log lines (total 73 in the set of 300) from *Hadoop*, *Zookeeper* and *Cassandra*, respectively, are discussed or inquired about through other mediums other than the project mailing list. Online issue reports (e.g., the Apache JIRA web interface) and *Stack Overflow* are the two sources where the log lines are discussed or inquired the most.

We also manually browsed the top 100 most frequently viewed questions on *Stack Overflow* for the tags "Hadoop" and "Cassandra" and we found 7 and 2 questions that inquired about log lines, respectively. For the tag "Zookeeper", there were only 15 questions in total with one of them being a log related inquiry. In short, practitioners inquire about logs

TABLE II: Types of information that practitioners asked about logs in the email inquiries.

Type	Explanation	Example
Meaning	Better description of the meaning of a log line is often sought.	An inquiry for a log line in *Hadoop* asks, "What exactly does this message mean?"
Cause	A clarification of the cause of a log line is commonly sought.	An inquiry for a log line in *Zookeeper* asked, "Does anybody know why this happened?"
Context	Practitioners often inquiry about the scenarios when a log line is printed.	An inquiry for a log line in *Cassandra* asked, "when does this occur?"
Solution	The inquiry seeks a solution for avoiding a particular log line, when the log line indicates an error.	An inquiry for a log line in *Hadoop* asked "It will be great if some one can point to the direction how to solve this".
Impact	The inquiry seeks the impact of a log line (e.g., whether a log line implies performance degradations).	An inquiry for a log line in *Cassandra* asked, "Is it affecting my data?"

with such inquiries being scattered across various mediums – potentially making them harder to archive and retrieve. Development knowledge often contains the answers for such inquiries.

III. Our Approach

In this section, we present the motivation and propose an automated *on-demand* approach to associate development knowledge to log lines.

Motivation for our approach: Only less than 1% of the log lines are inquired in the user mailing lists and other online sources. Hence, it's not cost-effective to document every log line. After examining the 15 lines in more detail we find that they have 5 different logging levels, the degree of fan-in among the methods that contain the logging statement varies from 0 to 2, and the number of changes to the methods that contain the logging statement varies from 1 to over 200. Since, there is nothing in common between the inquired log lines, it is difficult to even predict which lines to add documentation to by the developers. However, it still is a problem that needs to be solved, since real customers have issues with respect to understanding log lines as is evident from the emails to the mailing lists. Hence, in the rest of this section we propose an *on-demand* approach that will gather the required information when the practitioners need it with minimal intervention from the developers. Figure 1 shows a general overview of our approach. To ease explanation, we will use the log inquiry mentioned in Section I as an illustration example while presenting our approach (shown in Figure 2).

A. Extracting high-level source change information

J-REX [26] is used to extract historical information from Java software systems. We use J-REX to extract high-level source change information from the SVN repositories. J-REX extracts source code snapshots for each Java file revision from the SVN repository. Along with the extraction, J-REX also keeps track of all commit messages and issue report IDs in the commit messages for all revisions. For each source code snapshot of a file, J-REX builds an abstract syntax tree for the file using the Eclipse JDT parser.

B. Identifying logging statements

Software projects typically leverage logging libraries to generate logs. Typically, the logging source code contains method invocations that call the logging library. For example, in *Hadoop*, a method invocation with a caller "*LOG*" is considered as a logging statement in the source code. Knowing the logging library of the subject system, we analyze the output of J-REX to identify the code snippets responsible for logging, the corresponding historical changes to these logging statements and the source code in the method containing the logging statement. Some systems wrap the logging statements in other methods. For example, the inquired log line shown in Figure 2 is printed by a method "failedTask", which wraps the logging statement. Having the abstract syntax tree of the source code enables us to identify such methods that print log lines by wrapping logging statements.

C. Generating log templates

Typically a logging statement contains both static parts and dynamic parts. The static parts are the fixed strings and the dynamic parts are the variables whose values are determined at run-time. For example, in a logging statement *LOG.info("Retrying connect to server: Already tried" + n+ "time(s)"*, the variable n is the dynamic part, while *"Retrying connect to server: Already tried"* and *"time(s)"* are the static parts. We identify the static and dynamic parts in each logging statement and create a regular expression for each logging statement [10]. The regular expression generated for the above example is *Retrying connect to server: Already tried.*time\(s\)*. For the example shown in Figure 2, the regular expression generated for the logging statement is *.*Too many fetch-failures.**.

D. Associating development knowledge to log lines

Having all the regular expressions for the templates of logging statements, we are able to match log lines in the execution log files to the logging statements. For each of the inquired log lines, we find all the log templates that match it. We associate five sources of development data to the inquired log lines. We broadly categorize these five sources of development knowledge in to two categories:

Category 1 - Snapshot data: Snapshot knowledge includes information from the most up-to-date snapshot of the source code associated with a log line.

Source code: The source code in the method that contains the logging statement may provide information about the log line. For the example in Figure 2, by tracking the logging statement, we found that it is in the method "fetchFailureNotification" in file "JobInProgress.java".

Code comments: Sometimes the source code is not self-explanatory. In these cases, the code comment may be used to explain the source code and the associated logging statements. For the example in Figure 2, we find the code comment right

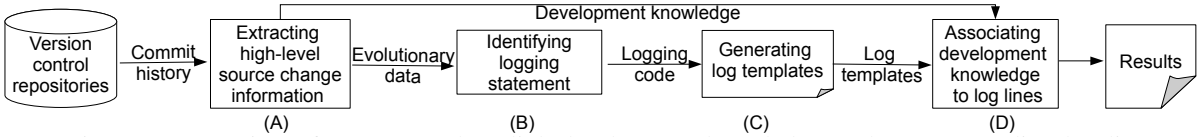


Fig. 1: An overview of our approach to link development knowledge to the corresponding log lines

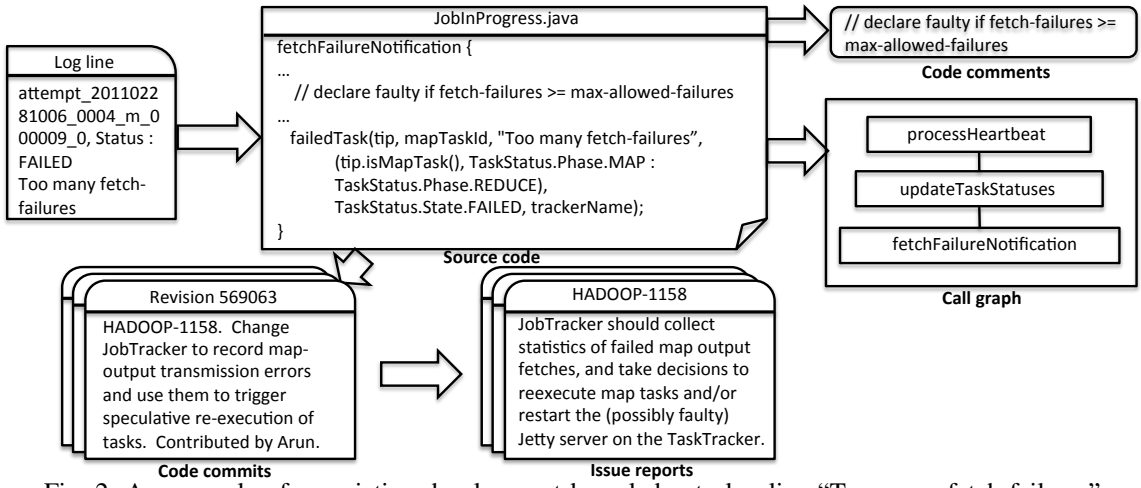


Fig. 2: An example of associating development knowledge to log line "Too many fetch-failures".

before the method saying "declare faulty if fetch-failures >= max - allowed-failures".

Call graph: Often, the logs only describe what happened instead of why the event happened or in what circumstances the event happened. When the reason or the context of a log line is sought out, the answer may be in the methods that trigger the method containing the logging statement. From the call graph of the log line in Figure 2, we find this method is called by method "processHeartbeat" and "updateTaskStatuses". Due to the complexity and size of the entire call graph, we only include the direct callers and callees of the method that containing the logging statement.

Category 2 - Historical data: Historical knowledge consists of the information that is generated during the development of the logging statement behind a log line or its associated source code. A previous study by Yuan *et al*. [27] shows the high churn rate of logging statements. The more a logging statement churns, the more historical data we have for the logging statement, i.e., we can learn more knowledge about logs from the development.

Code commits: A code commit contains the changes to the code and other corresponding information, such as the check-in comment describing the change and the developer who made the change. For example, the check-in comment for the change that adds or modifies a logging statement (or its surrounding or triggering code, calculated via the call graph) may provide information about the meaning of the log line. For the log line in Figure 2, we generate all 9 commits that change this method. One code commit (revision 569063), which introduce the log line, has message "*HADOOP-1158. Change JobTracker to record map-output transmission errors and use them to trigger speculative re-execution of tasks. Contributed by Arun.*". Another code commit (revision 606267) has message "*HADOOP-2247. Fine- tune the strategies for killing mappers and reducers due to failures while fetching map-outputs. Now the map- completion times and number of currently running reduces are taken into account by the JobTracker before killing the mappers, while the progress made by the reducer and the number of fetch-failures vis-a-vis total number of fetch-attempts are taken into account before the reducer kills itself. Contributed by Amar Kamat.*"

Issue reports: The source code changes are often due to issues (such as new feature requests and bugs), in the system. These issues are tracked in issue tracking systems, like JIRA. The report of an issue consists of its description, its resolution and developer discussions about it. An issue that is related to logs may be helpful in explaining the rationale of the logs. From the commit message attached to the inquired log line in Figure 2, we found issue reports with id "HADOOP-2247" and "HADOOP-1158".

We synthesizes the above data (i.e., source code, code comments, call graph, code commits and issue reports) into the following five types of development knowledge for the inquired log line:

- **Meaning:** The task does not respond for a number of times.
- **Cause:** Configuration mistake or Jetty issue are two of the possible reasons.
- **Context:** The event happens during the shuffle period.
- **Impact:** The event impacts the jobtracker component and will kill the corresponding task trackers.
- **Solution:** Updating configuration options or change Jetty versions may solve the issue.

IV. Can development knowledge help resolve real-life inquires?

In this section, we will apply our *on-demand* approach using three open source systems: *Hadoop*, *Cassandra* and *Zookeeper* (shown in Table I) in two steps. First, we examine whether our approach can resolve the real-life inquires identified in Section II. Second, we evaluate our approach against a large sample of log lines from the source code of the three subject systems.

A. Resolving real-life inquiries

In Sections II, we have found a set of real-life inquiries about log lines from mailing lists, issue reports and Stack Overflow. In this subsection, we would like to evaluate whether our approach can assist in resolving these real-life inquiries.

1) Approach: For the 15 log lines inquired in the user mailing list and 73 web search hits in total, we only focus on 14 log lines from the user mailing list and 31 web search results where we can clearly identify the specific inquired types of knowledge. Although some log lines are mentioned in the email, issue reports and other places on the web, we cannot easily identify the inquired knowledge.

For each of the 45 online inquiries (i.e., mailing list questions or search hits), we identify all the types of inquired information since each online inquiry may contain inquiries to multiple items of information. For example, inquiry # 1 in *Hadoop* asks about both the meaning and the cause of a log line (see Table III). In total, 21 items are inquired by the 14 email inquiries and 39 items are inquired by 31 web search inquiries. We then examine whether the various sources of development knowledge can resolve the inquiries.

2) Results: Our approach can be used to resolve real-life inquiries. Development knowledge provides answers to 9 out of 14 inquiries from the user mailing list. Table III shows that our approach can resolve 13 out of the 21 items of inquired information in the 14 real-life inquiries from the user mailing list. Issue reports are the best source of data to resolve real-life inquiries. Issue report can alone provide answers to 8 out of 9 log-inquiries from the mailing list that are resolved by our approach. In addition, 12 out of 13 resolved inquiry items from the mailing list are resolved by issue reports. Our approach provides answers to 15 out of 31 inquiries from the web search hits and resolves 16 out of 39 items of requested information. Each source of data performs similarly in resolving inquired items from the web search hits. Source code, code comment, call graph, commit and issue report can respectively resolve 4, 6, 3, 5 and 6 items of requested information from the web search hits.

3) Discussion: We discuss alternative approaches, such as web searching and reading the mailing list, to resolve real-life inquiries in this subsection.

Using the web search engine to resolve the real-life inquiries: We first compare the use of our approach and the use of a web search engine to resolve the real-life inquiries. One may consider using a web search engine, such as *Google*, to resolve log line inquiries. We use *Google* to search for the real-life inquired log lines and check whether the first 10 results from the web search engine can answer the inquiries. If the log lines can be ambiguous to other general terms, we add the name of the subject system after the log line.

We focus on the 14 real-life inquired log lines from the user mailing list since the other 31 real-life log lines are also from *Google* web search. For the 14 mailing list inquiries, we find four types of relevant search results from *Google*: the online link to the mailing list, the online link to the development knowledge (e.g., the Apache JIRA web interface), open source community websites (e.g., *Stack Overflow*) and personal websites. We find three inquired log lines where open source community websites can provide useful information and two inquired log lines where personal websites can provide useful information. From such results, we consider that the development knowledge (9 out of 14 inquired log lines) outperforms the results from a web search engine (5 out of 14 inquired log lines).

Using mailing list to resolve the real-life inquiries: We compare the use of our approach and the answers from mailing list to resolve the real-life inquiries. We find that our approach is comparable to the answers in the mailing list. In the mailing list, 10 inquiries are resolved after email discussion, while 9 inquiries are answered by development knowledge. However, we notice that the answers from mailing list are clearer and precise; while our approach returns more content and needs interpretation to resolve the inquiries. Nevertheless, our approach is *on-demand* and *automatic*. Hence the users can get the development knowledge almost instantly when using our approach, while they have to wait (sometimes as much as 105 hours) for someone to reply.

Our approach can provide help in resolving 9 out of 14 real-life inquiries from the user mailing list and 15 out of 31 real-life inquiries from other sources on the web. Development knowledge outperforms the simple web searches and is comparable to browsing the mailing list in resolving log line inquiries.

B. Complementing case study of randomly-sampled log lines

In the previous subsection, we have demonstrated that our approach can assist in resolving real-life inquiries. In this subsection, we want to check whether our approach can help understand a set of randomly-sampled log lines.

1) Approach: We select a random sample of 100 logging statements from all the logging statements throughout the lifetime of each of the subject systems. We then determine which of the 5 different types of inquired information (i.e., meaning, cause, impact, context and solution (see Section II)) by practitioners is missing from the log line itself. We then check whether our approach can complement the log lines using the development knowledge and fill in the missing information.

2) Results: We find that our approach of using the development knowledge can complement log lines by providing data on the 5 most common types of information inquired by users. As shown in Table IV, our approach can provide most of the missing meaning and context (on average 93% and 87%, respectively), around half of the missing cause and impact (on average 53% and 41% respectively) and only on average 12% of the missing solution. These percentages are calculated using the number of log statements with that type of development knowledge missing. We think the reason for such results is that, intuitively the meaning and context of the log line are the easiest types of information to get; the cause and impact of the log line are more difficult to find; while solving a logged error is the most difficult to uncover.

In particular, we find that issue reports are the source of data that can resolve the most log inquiries (see Table V). Issue reports are able to provide the four most-missing types of development knowledge (cause, context, solution and impact). Although source code is a source of data in providing the most meaning of log lines, most log lines have already contain the knowledge of meaning.

TABLE III: Result of using our approach to resolve the 14 real-life mailing list inquiries. Each table cell indicates the source of data that resolves the inquired information. A table cell with "not answered" indicates that the inquired knowledge is not provided by our approach. A blank cell indicates that the corresponding information was not inquired in the email. The first inquiry of Zookeeper did not request any specific information in the email and hence excluded in this table.

Hadoop	meaning	cause	context	solution	impact
1	call graph, issue report				
2		commit, issue report			
3		not answered		not answered	
4		issue report		issue report	
5		not answered			
6	commit, issue report	commit, issue report			
7		not answered		not answered	
8				not answered	
9		not answered			
10		source code, issue report		issue report	
Cassandra	**meaning**	**cause**	**context**	**solution**	**impact**
1		code comment, issue report	issue report		
2				not answered	source code
Zookeeper	**meaning**	**cause**	**context**	**solution**	**impact**
1		issue report			
2		issue report			

TABLE IV: Percentage of the logging statements complemented by our approach.

	meaning	cause	context	solution	impact
Hadoop	90% (19/21)	58% (57/98)	97% (69/71)	13% (13/99)	44% (40/91)
Cassandra	100% (21/21)	49% (48/98)	86% (65/76)	6% (6/100)	40% (36/90)
Zookeeper	89% (17/19)	51% (47/93)	79% (53/67)	18% (17/97)	39% (33/84)
Average	93% (57/61)	53% (152/289)	87% (187/214)	12% (36/296)	41% (109/265)

TABLE V: Number of log lines where each source of data can provide a particular type of development knowledge. The largest numbers in each type of knowledge are shown in bold.

Hadoop					
	meaning	cause	context	solution	impact
source code	**83**	12	23	1	5
comment	55	11	37	1	15
call graph	2	25	66	0	3
commit	42	26	61	4	11
issue report	49	**37**	**69**	**12**	**27**
Cassandra					
	meaning	cause	context	solution	impact
source code	**59**	4	14	0	11
comment	47	19	26	1	**18**
call graph	0	11	39	0	1
commit	43	17	45	1	5
issue report	47	**24**	**51**	**5**	15
Zookeeper					
	meaning	cause	context	solution	impact
source code	**59**	9	12	0	13
comment	41	18	30	4	17
call graph	2	6	40	0	0
commit	37	22	45	3	5
issue report	51	**34**	**56**	**17**	**25**

3) Discussion: We discuss how each type of inquired information assists in understanding log lines.

Meaning. The meaning of a log line is inquired because the text in the log line is not descriptive enough. As we can see from Table V, source code is the best source for getting the meaning of a log line. For example, a log line from *Hadoop* prints "-files", which does not have a clear meaning. From the source code, we can find out that the log line corresponds to temporary files defined by a user from command line. Issue reports also provide useful information about the meaning of log lines. For example, the *Hadoop* issue report "HADOOP-182" is associated with the log line "lost tracker". In the discussion for the issue report, the meaning of the log line was clearly presented as: "When a Task Tracker is lost (by not sending a heartbeat for 10 minutes), the JobTracker marks the tasks that were active on that node as failed". Therefore, we can say that the "lost" message in the log line means that a heartbeat from the tracker was not received.

Cause. 11 out of 15 email inquiries in Section II asked about the cause for a log line. Our results in Table V show that both sources of historical development knowledge, i.e. commit messages and issue reports, can help in explaining the reason of some log lines. Issue reports are the source of development knowledge in providing the cause of log lines in most cases. There are typically two scenarios when an issue report would provide such useful information:

1) A logging statement is added into the source code as part of a feature or improvement. For example, to resolve *Hadoop* issue "HADOOP-1171" (where the issue was to enable multiple retries of reading the data), log line "fetch failure" was added.

2) A logging statement is explained in the discussion of the issue report when a bug in the source code is identified. For example, in the discussion about *Hadoop* issue "HADOOP-1093", the reason for log line "NameSystem.completeFile: failed to complete" in *Hadoop* was explained as, a race condition between a client and data server of *Hadoop*. The commit message is another source of development knowledge for explaining the cause of a log line. For example, a log line "DIR * FSDirectory.mkdirs: failed to create directory" was added in revision 412474 of *Hadoop*, with a commit message

TABLE VI: Resolved and un-resolved email inquiries.

System	total	resolved		not-resolved		
		by expert	by non-expert	replied by expert	only replied by non-expert	not replied
Hadoop	10	5	1	0	1	3
Cassandra	2	0	2	0	0	0
Zookeeper	3	3	0	0	0	0

"Fix DFS mkdirs() to not warn when directories already exist", which clearly indicates the cause.

Impact. Source code can provide the information about which components contain the event causing the log line. For example, from the source code of log line "initialization failed", we find that the logging statement is embedded in the constructor method of class "FSNamesystem". Therefore, we have the information that the initialization failure is in the name index component of the file system. The source code can also provide information about other components that might be impacted by the event causing the log line to be printed. Note that, although some logging libraries, such as Log4j, can provide the information about the component that generates the logs, information about other indirectly impacted components cannot be gathered using logging library.

Context. Context of the log line is important in understanding log lines. Call graph is one type of development knowledge that provides context for the log lines. For example, the call graph of log line "Column family ID mismatch" of *Cassandra* tells that the log line is in the update period of the system. Most logging libraries provide a time stamp for each log line. However, such time stamps are not as useful as the domain-specific context. For example in MapReduce, the time stamp of a log line is not as useful as knowing if the log line is printed in the map period or reduce period.

Solution. If a log line indicates an error, one would want to know how to solve the error. However, since many complex reasons could potentially cause one error, log lines typically do not contain information on how to solve the error. It is difficult to provide a solution to a logged error. Prior research by Thakkar *et al*. [28] proposes a technique to find similar customer engagement reports to solve field issues. Similarly, issue reports can provide some solutions for logged errors. We find cases where developers discuss the log-related issues in the issue tracking system. For example, the log line "Severe unrecoverable error, exiting" from *Zookeeper* is related to issue "ZOOKEEPER-1277", where a developer describes the way to resolve this issue by patching to a new version.

Our approach can assist in complementing the information available in log lines by using development knowledge. In particular, issue reports are the best source to provide the inquired information for log lines, among other sources of data.

V. Can Experts Assist in Resolving Inquiries of Log Lines?

Logging statements are embedded in the source code intentionally by developers. Intuitively, the developers who added the log lines may be the most suitable person to address any log line inquiries. From our manual examination, we find that the replies in the user mailing lists of the subject systems often resolve the inquiries (see Section II). We would like to verify whether experts of the log lines, such as the developers who added the log line, do provide such information more often than non-experts of the log lines. If the experts of the log lines can better provide extra information about log lines, our development knowledge could help on two aspects. First, we can use development knowledge and leverage existing techniques [29] to identify the experts of the log lines to rapidly re-route log line inquiries to the most appropriate developer (i.e., the owner of the log line). Second, in the case of multiple replies or posts for a particular question or web search, we can use development knowledge to identify expert replies or posts.

A. Approach

Previous research by Mockus *et al*. [29] proposed a technique that uses the code change information to identify the experts of code units, such as modules and methods. Based on this technique, we define an expert for a log line as *"a developer who has committed changes to the method/function that contains the logging statement, which generates the log line."* For example, log line "Lost tracker" of *Hadoop* is in a method called *"lostTaskTracker"* and all the 10 developers who have committed changes to the method "lostTaskTracker" are considered as log experts for that log line.

We read the email threads of all the 15 inquired log lines and assign them to 1 of 5 categories: resolved by expert, resolved by non-experts, not-resolved but replied by expert, not-resolved but only replied by non-expert and not replied. We also calculate the 'time for first reply' for experts and non-experts to measure their response time.

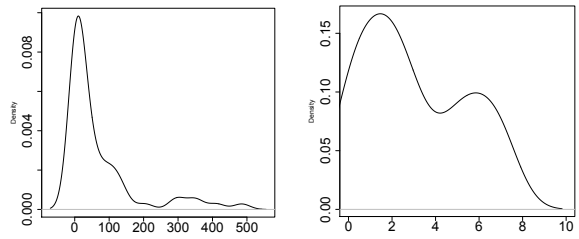
B. Results

We answer the following two questions:

1. Who resolved the inquiries in the email threads? An overview of the results is presented in Table VI. We find that an expert is crucial in providing answers to log line inquiries – 8 of the 11 resolved inquiries had replies by log experts. The one inquiry from *Hadoop* that was tagged as "resolved by non-expert" was actually resolved by the person who had the inquiry. The two inquiries resolved by non-expert from *Cassandra* were resolved by other developers of the projects. We find that experts have considerable knowledge about the log line itself, as well as the rest of the project which assists them in resolving inquiries of log lines. For example, an expert pointed out that an inquired log line from *Zookeeper* was due to a fixed bug. It is hard for a non-expert to provide such information.

It is also interesting to observe that all the inquiries that are answered by experts are done through short and affirmative answers. For example, the email that inquired about "Lost tracker" was replied to by one expert and one non-expert. The non-expert asked about the context of the log line, while the expert directly gave the answer and also pointed out that there were bugs related to this log line. We also note that the experts for all the inquired log lines are still active members in the development team of the three subject systems. Therefore the log line inquiries that had no replies were not due to the absence of experts in the development team, but most probably due to the experts not being aware of such inquiries.

2. How long did the first reply in the email thread take? The median times for the first reply from experts in *Hadoop* and



(a) number of emails (b) number of years

Fig. 3: Density plot of the number of emails and active years of the experts of the inquired log lines from Hadoop.

TABLE VII: 'Time for first reply' of email inquiries.

	Expert			Non-expert		
	Median	Min	Max	Median	Min	Max
Hadoop	3 h	3 min	96 h	1 h	8 min	2 h
Cassandra	-	-	-	7 min	77 h	105 h
Zookeeper	5 h	1 h	9h	-	-	-

Zookeeper are 3 and 5 hours respectively. However, the experts from *Cassandra* never replied (see Table VII). The 'time for first reply' from experts is sometimes slower than non-experts. From browsing the email replies, we find the reason for slower reply is that experts often reply with a definitive answer, while non-experts often ask for more information. Therefore, although experts might reply later than non-experts, they often provide more useful information than the non-experts.

Our findings provide evidence that experts are important in providing useful information about log lines. Therefore, finding the expert of a log line may be the most effective way for understanding the log line. However, a person with a log line inquiry won't be able to direct their inquiries to experts since information about log line expertise is not easily accessible. Moreover, experts might be too busy to check the mailing list. Automated ways to push inquiries to experts might be of great value for understanding log lines.

C. Discussion

Experts can provide useful information about a log line. However, we do not know how many experts are there for a particular log line, how many of these experts are active on the mailing lists and how many orphaned log lines exist without any experts. Since these questions shed more light on the experts of a log line, we decided to answer them. The results for these questions are below.

1. How many experts are there for the inquired log lines?We manually check the experts of the 300 sampled logging statements in the 3 subject systems. We not only check the developer who commits the revisions, but we also read the commit comments. If a commit comment says that the revision is contributed by another developer or that the developer has committed the revision on behalf of another developer, then we consider the developer who coded the revision as the expert instead of the developer who committed the revision.

We find that on average, there are 4.6, 3.1 and 2.8 experts for each log line in *Hadoop*, *Cassandra* and *Zookeeper*, respectively. We also find that some log lines from *Hadoop* have over 30 experts since the log lines are embedded inside large methods, while some log lines only have 1 expert. It would be easier to find an expert for the log lines in large methods since there are more experts to answer the inquiries; while for the log lines with only 1 expert, if the single expert is not available or not currently working as part of the team, it would be hard to find someone to answer the inquiry about these log lines.

2. How active are experts on the mailing list? We examine all the experts of the 15 inquired log lines in the mailing list. For each expert, we count the number of emails sent and the number of active years in the mailing list. Figure 3(a) shows the distribution of the number of emails sent by the experts of the 10 inquired log lines for *Hadoop*. We can identify three types of experts: 1) heavy email contributors, who might send hundreds of emails over the years and are typically experts of the entire project, 2) medium email contributors, who would reply to the inquiries about which they have knowledge and 3) light email contributors who are typically not considered in the "core" of the development team.

From the number of active years shown in Figure 3(b) for the log-line experts in *Hadoop*, we observe that some experts are active in the mailing list throughout the entire history of the project (which is 8 years), while other ones are only active for a short period of time.

We also find that in the expert-resolved inquiries, the inquiries are resolved by the experts who are heavy email contributors with long active years. This finding also indicates the value of automatically pushing inquiries to experts since not all experts might be active on the mailing list and are often likely to miss inquiries, which they could easily resolve.

3. How many orphan log lines are there with no experts? We define orphan log lines as the log lines whose experts are no longer committing code changes to the project since the last release to the time of the inquiry. We examine all experts of the 15 log lines that are inquired in the mailing lists and we find that none of the log lines are orphaned. Such a result indicates that looking for the experts of log lines may be one of the best ways to resolve inquiries about log lines since it is unlikely that the inquired log line is an orphan.

However, as we can see in Tables VI and VII, there are cases where there are no replies or the replies take as much as 4 days. The person who has posed the inquiry might need information much faster, especially when the log line is associated with an error. Hence, it would be beneficial if one could either directly contact the log-line expert or automatically get the same information provided by the experts elsewhere and without having to wait long.

The responses from experts of logs can assist in log line understanding. Development knowledge derived from our approach can help identify experts.

VI. Related Work

Documenting Code. There are many prior research studies focusing on providing documentation to assist in the better understanding of source code. Haiduc *et al*. [30], [31] used automated text summarization techniques, such as *LSI* and *VSM* to create summarized descriptions for source code. They performed an user study to evaluate the quality of the automatically generated summarization. Sridhara *et al*. [32] describe a technique that automatically generates comments for Java methods. Their technique focuses on describing the roles of parameters in a method. Padioleau *et al*. [33]

performed a qualitative study on the source code comments of three large open source operating systems and found that code comments can complement software documentation and record the thoughts of developers during the development in an informal manner. Padioleau *et al*. [33] claimed that source code comments, an important source of documentation, contains valuable knowledge from developers. They performed a qualitative study on the source code comments from three large open source operating systems and found that code comments can complement software documentation and record the thoughts of developers during the development in an informal manner. Instead of improving understandability of source code, our work focuses on assisting understanding log lines that is widely used in software maintenance, yet typically with poor documentation.

Gathering knowledge of IT events. Prior research has presented an approach that gather knowledge of IT events [34]. The approach searches the IT events from online data sources, such as *Oracle online forum*. The search results, i.e. a list of documents relevant to IT events, are enhanced by a ranking system based on content source and quality of information relevancy. Our approach considered another source of knowledge, i.e., the development knowledge. From our evaluation shown in Section IV, our approach outperforms online search results and email threads.

Improving log lines. There are prior research studies that aim to automatically improve the information in log lines by providing the context and solution.

1. Providing context. Some logging libraries, such as Log4j, can automatically output the name of the class, in which the log is generated. Such information is useful to understand the context of the logs. Yuan *et al*. [5] propose an approach to automatically enhance logging statements by printing the values of the accessible variables. This technique assists in adding more context to the log lines. Shang *et al*. [35] design a technique to automatically provide context information for log lines to assist in deploying applications in a cloud environment. Beschastnikh *et al*. [36] build models from logs to infer the state of a system when an error occurs.

2. Providing solution. Ding *et al*. [37] designed a framework to correlate logs, system issues and corresponding simple solutions. They store such information in a database to assist in providing simple solutions, such as "rebooting an application", when similar logs appear.

Instead of improving the documentation of code or improving the information in the logs lines, our work focuses on assisting developers in understanding log lines that are widely used in practice, yet typically have poor documentation. Our work can potentially improve the existing log analysis techniques by providing better understanding of log lines.

VII. Threats to Validity

External validity. Our case study only focuses on 300 randomly sampled logging statements, 15 email log lines inquiries and 73 log line inquiries from the web hits for three "systems" projects with years of history and a large user base. The results might not generalize to other systems. Our case studies only focus on three Java systems. Our approach can be extended to support systems with other programming languages too. However, a specific parser needs to be implemented for for each programming language. Additional case studies on other open source and commercial systems in different domains are needed. We also plan to perform user studies to evaluate the usefulness of our approach. There might be other sources of development knowledge, such as IRC channels, design documents and code review comments, which maybe useful to understand log lines. Adding additional information could improve our approach. A broader study, which includes more sources of development knowledge needs to be conducted to identify the potential of each source in resolving the various types of log line inquiries.

Construct validity. The manual examination throughout the study is carried out by the authors of this paper showing our approach is fully accurate. Although we have some experience in using and studying the subject systems in our previous research (e.g., [35]), our expertise in them might be imperfect. We plan to have a larger user study to evaluate our approach.

We choose to attach development knowledge to log lines at the method level. The higher the level, the more development knowledge that can be attached, but the more overwhelming such attached knowledge might become. If we set the level lower (e.g., log statement level), we would lose some useful knowledge about log lines. We plan to explore attaching development knowledge at different levels of granularity.

Our approach treat all types of logging statement the same. However, different types of logging statements may require different information for improvement. We plan to improve optimize our approach by considering different types of logging statements. In our approach, we consider all code commits that change the logging statement as part of the development knowledge. However, some code commits may not provide useful information, such as renaming a variable. Filters can be applied to further improve our approach. Based on previous research of source code expertise [29], we consider developers who commit code that changes the methods containing the logging statements as the experts of the log lines. Such an assumption may not be entirely correct. There could be other approaches to refine the set of experts for log lines. Even with this naive approach, we are able to show the importance of experts in resolving log line inquiries.

VIII. Conclusion

Much of the knowledge about log lines lives only in the minds of the developers who embedded the logging statements in the source code. Due to the lack of communication with developers of large software systems, practitioners, who might heavily rely on logs for software maintenance tasks, often face the challenge of understanding the logs. Such challenges may jeopardize the effectiveness and correctness of leveraging logs.

To address the challenge of understanding log lines, we propose an systematic on-demand approach to recover development knowledge for log lines. To evaluate our approach, we studied three open source systems: *Hadoop*, *Cassandra* and *Zookeeper*. In our case study, we manual examine 300 randomly sampled logging statements from the subject systems and use 45 real-life inquiries in the user mailing lists of the three subject systems and from web search hits. We find that:

- Practitioners sometimes find it difficult to understand the log lines. The five types of information (i.e., cause,

meaning, impact, context and solution) about the log lines are inquired in the mailing lists.
- Development knowledge can be leveraged for understanding log line. In particular, issue reports are useful for log line understanding.
- Our approach can also be used to identify experts of a particular log line.

In conclusion, our study demonstrates the importance and opportunities of leveraging development knowledge for understanding log lines.

REFERENCES

[1] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "Leveraging performance counters and execution logs to diagnose memory-related performance issues," in *ICSM '13: Proceedings of the 2013 IEEE International Conference on Software Maintenance*, 2013, pp. 110–119.

[2] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *ICSM '09: 25th IEEE International Conference on Software Maintenance*, 2009, pp. 125–134.

[3] ——, "Automatic identification of load testing problems," in *ICSM '08: Proc. of 24th IEEE International Conf. on Software Maintenance*, 2008, pp. 307–316.

[4] W. Shang, M. Nagappan, and A. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, pp. 1–27, 2013.

[5] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *ASPLOS '11: Proc. of the 16th international conference on Architectural support for programming languages and operating systems*, 2011, pp. 3–14.

[6] L. Bitincka, A. Ganapathi, S. Sorkin, and S. Zhang, "Optimizing data analysis with a semi-structured time series database," in *SLAML'10: Proceedings of the 2010 workshop on Managing systems via log analysis and machine learning techniques*, 2010, pp. 7–7.

[7] "Xpolog," http://www.xpolog.com/.

[8] "logstash," http://logstash.net/.

[9] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *CCGRID '10: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 94–103.

[10] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *SOSP '09: Proc. of the ACM SIGOPS 22nd symp. on Operating systems principles*, 2009, pp. 117–132.

[11] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *ICDM '09: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, 2009, pp. 588–597.

[12] S. R. Sandeep, M. Swapna, T. Niranjan, S. Susarla, and S. Nandi, "Cluebox: a performance log analyzer for automated troubleshooting," in *WASL'08: Proceedings of the First USENIX conference on Analysis of system logs*, 2008, pp. 1–1.

[13] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, 2013.

[14] ——, "An exploratory study of the evolution of communicated information about the execution of large software systems," in *WCRE '11: Proc. of the 18th Working Conference on Reverse Engineering*, 2011, pp. 335–344.

[15] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *OSDI'12: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, 2012.

[16] A. E. Hassan and R. C. Holt, "Using development history sticky notes to understand software architecture," in *IWPC 2004: Proc. of the 12th IEEE Int. Workshop on Program Comprehension*, 2004.

[17] D. Čubranić and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 408–418.

[18] E. L. A. Baniassad, G. C. Murphy, and C. Schwanninger, "Design pattern rationale graphs: linking design to source," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 352–362.

[19] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 406–416.

[20] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *ICSE '10: Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, 2010, pp. 375–384.

[21] B. Dagenais and M. P. Robillard, "Recovering traceability links between an API and its learning resources," in *ICSE 2012: Proceedings of the 34rd ACM/IEEE International Conference on Software Engineering*, June 2012, pp. 47–57.

[22] C. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Eng.*, vol. 25, no. 4, pp. 557 –572, Jul/Aug 1999.

[23] "Sans consensus project - information system audit logging requirements," http://www.sans.org/security-resources/policies/info_sys_audit.pdf.

[24] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé, "Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations," in *AOSD '07: Proc of the 6th International Conference on Aspect-oriented software development*, 2007, pp. 199–211.

[25] A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield, and D. Dietz, "An Industrial Case Study of Customizing Operational Profiles Using Log Compression," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*, 2008, pp. 713–723.

[26] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan, "MapReduce As a General Framework to Support Research in Mining Software Repositories (MSR)," in *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 21–30.

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

[28] D. Thakkar, Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora, "Retrieving relevant reports from a customer engagement repository," in *ICSM '08: Proc. of the 24th IEEE International Conference on Software Maintenance*, 2008, pp. 117–126.

[29] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *ICSE '02: Proc. of the 24th International Conference on Software Engineering*, 2002, pp. 503–512.

[30] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *WCRE '10: Proceedings of the 2010 17th Working Conference on Reverse Engineering*, 2010, pp. 35–44.

[31] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *ICSE '10: Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, 2010, pp. 223–226.

[32] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *ICPC '11: Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 71–80.

[33] Y. Padioleau, L. Tan, and Y. Zhou, "Listening to programmers - Taxonomies and characteristics of comments in operating system code," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 331–341.

[34] G. Barash, I. Cohen, E. Mordechai, C. Staelin, and R. Dakar, "Creating the knowledge about IT events," in *SLAML'10: Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, 2010.

[35] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 402–411.

[36] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *ESEC/FSE '11: Proc. of the 19th ACM SIGSOFT symp. and the 13th European conf. on Foundations of software engineering*, 2011, pp. 267–277.

[37] R. Ding, Q. Fu, J.-G. Lou, Q. Lin, D. Zhang, J. Shen, and T. Xie, "Healing online service systems via mining historical issue repositories," in *ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 318–321.