An Exploratory Study of the Evolution of Communicated Information about the Execution of Large Software Systems

Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan
School of Computing
Queen's University
Kingston, Ontario, Canada
{swy, zmjiang, bram, ahmed}@cs.queensu.ca

Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
migod@uwaterloo.ca

Mohamed Nasser, Parminder Flora
Performance Engineering
Research In Motion (RIM)
Waterloo, Ontario, Canada

***Abstract*—A great deal of research in software engineering focuses on understanding the dynamic nature of software systems. Such research makes use of automated instrumentation and profiling techniques after fact, i.e., without considering domain knowledge. In this paper, we turn our attention to another source of dynamic information, i.e., the Communicated Information (*CI*) about the execution of a software system. Major examples of *CI* are execution logs and system events. They are generated from statements that are inserted intentionally by domain experts (e.g., developers or administrators) to convey crucial points of interest. The accessibility and domain-driven nature of the *CI* make it a valuable source for studying the evolution of a software system. In a case study on one large open source and one industrial software system, we explore the concept of *CI* and its evolution by mining the execution logs of these systems. Our study illustrates the need for better traceability techniques between *CI* and the *Log Processing Apps* that analyze the *CI*. In particular, we find that the *CI* changes at a rather high rate across versions, leading to fragile *Log Processing Apps*. 40% to 60% of these changes can be avoided and the impact of 15% to 50% of the changes can be controlled through the use of the robust analysis techniques by *Log Processing Apps*. We also find that *Log Processing Apps* that track implementation-level *CI* (e.g., performance analysis) are more fragile than *Log Processing Apps* that track domain-level *CI* (e.g., workload modeling), because the implementation-level *CI* is often short-lived.**

***Index Terms*—Reverse engineering, Software evolution, Communicated information, Execution log analysis**

I. Introduction

Software profiling and automated instrumentation techniques are commonly used to study the run-time behaviour of software systems [1]. However, such techniques often impose high overhead and slow down the execution, especially for real-life workloads. Worse, software profiling and instrumentation are done after the fact with limited domain knowledge. Therefore, massive instrumentation often leads to an enormous volume of results that are impractical to interpret meaningfully.

In practice, system administrators and developers typically rely on the software system's Communicated Information (*CI*), consisting of the major system activities (e.g., events) and their associated contexts (e.g., a time stamp), to understand the high-level field behaviour of large systems and to diagnose and repair bugs. Rather than generating tracing information in a blind way, developers choose to explicitly communicate specific *CI* because they consider the information to be particularly important. Execution logs are one way to persistently store *CI*. The importance of the information varies based on the purpose of the logs. For example, detailed debugging logs are relevant to developers (i.e., implementation *CI*), while operation logs summarizing the key execution steps are more relevant to operators (i.e., domain-level *CI*).

The rich nature of *CI* has introduced a whole new market of applications that complement large software systems. We collectively call these applications the *Log Processing Apps*. The *Apps* are, for example, used to generate workload information for capacity planning of large-scale systems [2], [3], to monitor system health [4], to detect abnormal system behaviours [5], or to flag performance degradations [6].

Often, *Log Processing Apps* are in-house applications that are highly dependent on the *CI*. Although they are typically built on commercial platforms by IBM [7] and Splunk [8], the actually link between the *Log Processing Apps* and the monitored system is formed by the specific kind and format of *CI* in use. Hence, the *Apps* require continuous evolution as the *CI* changes and as the needs change. In our experience, *CI* changes often break the functionality of the *Log Processing Apps*. However, since little is known about the evolution of *CI*, it is unclear how much maintenance effort *Log Processing Apps* require in the long run.

In this paper, we explore the evolution of *CI* by examining the logs of 10 releases of an open source software system (*Hadoop*) and 9 releases of a closed source large enterprise application, which we call *EA*. Our study is the first step in understanding the maintenance effort for *Log Processing Apps* by studying the evolution of the *CI* (i.e., their input domain). We explore the following research questions by tracking the *CI* for a fixed set of major features across the life-time of the two studied systems:

RQ1: How does the *CI* evolve over time?

TABLE I
OVERVIEW OF THE STUDIED RELEASES OF *Hadoop* (MINOR RELEASES IN ITALIC)

Release	Release Date	K SLOC
0.14.0	20 August, 2007	122
0.15.0	29 October, 2007	137
0.16.0	7 February, 2008	181
0.17.0	20 May, 2008	158
0.18.0	22 August, 2008	174
0.19.0	21 November, 2008	293
0.20.0	22 April, 2009	250
0.20.1	14 September, 2009	258
0.20.2	26 February, 2010	259
0.21.0	23 August, 2010	201

Surprisingly, we find that over time the *CI* about these features increase by 1.5-2.8 times compared to the first-studied release. Across releases, we note that 20-30% of the *CI* is changed with context modifications across releases. These changes are troublesome as they cause the *Log Processing Apps* more error-prone.

RQ2: What types of modifications happen to *CI*?
Examining the 20-30% CI changes across releases, we identify six types of CI modifications. Of these changes, 40-60% can be avoided and the impact of 15-50% of them can be minimized through the use of robust analysis techniques. The remaining modifications are risky and should be tracked carefully to avoid breaking *Log Processing Apps*.

RQ3: What information is conveyed by the short-lived *CI*?
We find that short-lived *CI* contains implementation-level details. More resources should be allocated to maintain *Log Processing Apps* that heavily depend on implementation-level information.

Our study highlights the need for tools and approaches (e.g., traceability techniques) to ease the maintenance of *Log Processing Apps*.

The rest of this paper is organized as follows: Section II presents an example to motivate our work. Section III presents the data preparation steps for our case study. Section IV presents our case studies and the answers to our research questions. Section V discusses the limitations of our study. Section VI discusses prior work. Section VII concludes the paper.

II. A MOTIVATING EXAMPLE

We use a hypothetical motivating example to illustrate the impact of *CI* changes on the development and maintenance of *Log Processing Apps*.

The example considers an online file storage system that enables customers to upload, share and modify files. Initially, there were execution logs used by operators to monitor the performance of the system. The information recorded in the execution logs contained system events, such as "user request file", "start to transfer file" and "file delivered".

Release n

Operators identified a performance problem in the prior release. In order to diagnose the problem, developer Andy added more information to the context of the execution event in *CI*, such as ID of the thread that handles file uploading. Using the added context in the execution logs, the operators identified the reason of the performance problem and developer Andy resolved the problem. A simple *Log Processing App* was written to continuously monitor for the re-occurrence of the problem by scanning the *CI* for the corresponding system event.

Release n+1

The file upload feature was overhauled, leading the developers to change the communicated events and their associated logs. The context change to the log files led to failures of the *Log Processing Apps*. The application could not parse the log lines correctly to find the start and end time stamps of each transaction. The application started giving false alarms. After several hours of analysis, the root-cause of the false alarm was identified. The *CI* was changed by another developer Bob who was not aware (no traceability) that others made use of this information.

To avoid problems reoccurring in the future, developer Andy marked the dependence of the *Log Processing App* on this *CI* event in an ad hoc manner through basic code comments.

From the motivating example, we can observe the following:

- *CI* is crucial for understanding and resolving field problems and bugs.
- *CI* is continuously changing due to development and field requirements.
- *Log Processing Apps* are highly dependent on the *CI*.

Unfortunately, today there are no techniques to document such dependencies, leading *Log Processing Apps* to be very fragile as they adapt to continuously changing *CI*.

III. CASE STUDY SETUP

To understand how *CI* changes, we mine a commonly available source of *CI*: the execution logs. In this section, we present the studied systems and our approach to recover the *CI* from the execution logs.

A. Studied systems

We choose one open source system and one closed source software system with different size and application domain as the subject for our case study. We choose 10 releases of *Hadoop*, an open source software application[1], and 9 releases of a closed source large enterprise application (*EA*).

Hadoop is a large distributed data processing platform that implements the MapReduce [9] data processing paradigm. We use releases 0.14.0 to 0.21.0 for our study as shown in Table I). We choose these releases since 0.14.0 is the earliest one that we could deploy in our experimental environment and 0.21.0 is the most recent release at the time of this study. Among

[1]http://hadoop.apache.org/, last checked March 2011.

the studied releases, 0.20.1 and 0.20.2 are minor releases of the 0.20.0 series. Because *Hadoop* is widely used in both academia and industry, various *Log Processing Apps* (e.g., *Chukwa* [10] and Salsa [11]) are designed to diagnose system problems as well as monitor the performance of *Hadoop*.

The enterprise application (*EA*) in our study is a large-scale, communication application that is deployed in thousands of enterprises worldwide and used by millions of users. Due to a Non-Disclosure Agreement, we cannot reveal additional details about the application. We do note that it is considerably larger than *Hadoop* and has a much larger user base and longer history. We studied 9 releases of *EA*. The first 7 minor releases are from one major release series and the later 2 releases are from another major release. We name the release numbers 0.1.0 to 0.1.6 for the first major release and 0.2.0 to 0.2.1 for the second major release. There are currently several *Log Processing Apps* for the *EA*. These *Log Processing Apps* are used for functional verification, performance analysis, capacity planning, system monitoring, and field diagnosis by customers worldwide.

B. Uncovering CI from logs

Our approach to recover the *CI* of software systems consists of the following three steps: 1) System deployment, 2) Data collection, and 3) Log abstraction.

System Deployment

For our study, we seek to understand the *CI* of each system based on exercising a fixed set of features across the releases of these systems. To achieve our goal, we run every version of each application with the same workload in an experimental environment. The experimental environment for *Hadoop* consists of three machines. The experimental environment for *EA* mimics the setup of a very large field deployment.

Data collection

In this step, we collect execution logs from the two subject systems.

The Hadoop workload

The *Hadoop* workload consists of two example programs, *wordcount* and *grep*. The *wordcount* program generates the frequency of all the words in the input data and the *grep* program searches the input data for a string pattern. In our case study, the input data for both *wordcount* and *grep* is a set of data with a total size of 5 GB. The search pattern of the *grep* program in our study is a word ("fail").

The Enterprise System workload

To collect consistent and comparable data, we choose a subset of features that are available in all versions of the *EA*. We simulate the real-life usage of the *EA* system through a specialized workload generator, which exercises the configured set of features for the same number of times..

We perform the same 8-hour standard load test [12] on each of the *EA* releases. A standard load test mimics the real-life usage of the system and ensures that all of the features are covered during the test.

Log abstraction

We recover the *CI* by analyzing the generated execution logs. Execution logs (e.g., Table II) typically do not follow a strict format. Instead, they often use inconsistent formats [13]. The free-form nature of these logs makes it hard to extract information from them. Moreover, log lines typically contain a mixture of static and dynamic information. The static values contain the descriptions of the execution events, while the dynamic values indicate the corresponding *context* of these events.

We need to identify the different kinds of system events based on the different event instances in the execution logs. We use a technique proposed by Jiang *et al*. [14] to automatically extract the execution events and their associated context from the logs. Figure 1 shows the overall process of log abstraction. As shown in Table III, the descriptions of task types, such as "Trying to launch", are static values, i.e., system events. The time stamps and task IDs are dynamic values (i.e., the context for these events). The log abstraction technique normalizes the dynamic values and uses the static values to create abstracted execution events. We consider the abstracted execution events as representations of communicated system events.

First, the anonymize step uses heuristics to recognize dynamic values in log lines. For example, "TaskID=01A" will be anonymized to "TaskID=$id". The tokenize step separates the anonymized log lines into different groups (i.e., bins) according to the number of words and estimated parameters in each log line. Afterwards, the categorize step compares log lines in each bin and abstracts them into the corresponding execution events (e.g., "Reduce"). Similar execution events with different parameters are categorized together. Since the anonymize step uses heuristics to identify dynamic information in a log line, there is a chance that the heuristic might miss to anonymize some dynamic information. The reconcile step identifies such dynamic values by comparing the execution events in the same bin to each other. Case studies in previous research [14] show that the precision and recall of this technique both are high (i.e., over 80% precision and recall on the *EA* logs).

Performing static analysis on source code is another approach to abstract execution logs [15]. However, we do not use this approach because of two major reasons: 1) in practice, developers of *Log Processing Apps* typically do not have access to the source code, and 2) sophisticated static and dynamic slicing techniques are needed to determine the actual *CI* based on the code as a log line might be produced by several source code lines.

IV. Case Study Results

In this section, we present the findings on our research questions. For each research question, we present the motivation, our approach, and the corresponding results.

RQ1: How does the *CI* evolve over time?

Motivation

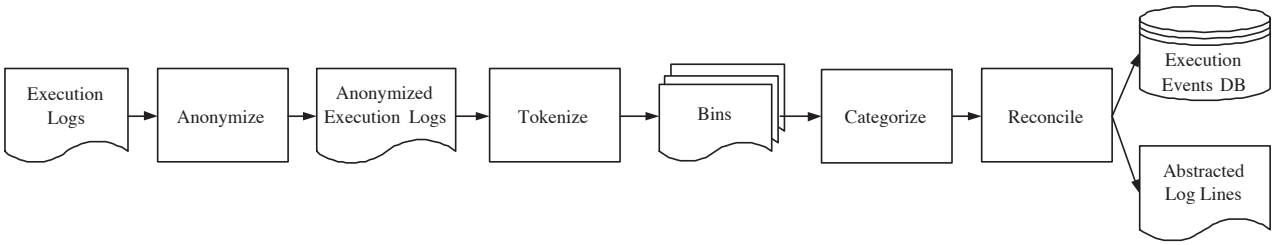

Fig. 1. Overall framework for log abstraction.

TABLE II
EXAMPLE OF EXECUTION LOG LINES

#	Log lines
1	time=1, Trying to launch, TaskID=01A
2	time=2, Trying to launch, TaskID=077
3	time=3, JVM, TaskID=01A
4	time=4, Reduce, TaskID=01A
5	time=5, JVM, TaskID=077
6	time=6, Reduce, TaskID=01A
7	time=7, Reduce, TaskID=01A
8	time=8, Progress, TaskID=077
9	time=9, Done, TaskID=077
10	time=10, Commit Pending, TaskID=01A
11	time=11, Done, TaskID=01A

TABLE III
ABSTRACTED EXECUTION EVENTS

Event	Event template	#
E_1	time=$t, Trying to launch, TaskID=$id	1,2
E_2	time=$t, JVM, TaskID=$id	3,5
E_3	time=$t, Reduce, TaskID=$id	4,6,7
E_4	time=$t, Progress, TaskID=$id	8
E_5	time=$t, Commit Pending, TaskID=$id	10
E_6	time=$t, Done, TaskID=$id	9,11

The evolution of *CI* impacts the maintenance of *Log Processing Apps*. The frequent change of the *CI* makes *Log Processing Apps* fragile due to the lack of established traceability techniques between *Log Processing Apps* and *CI*. Hence, change to execution events gives an indication of the complexity of designing and maintaining *Log Processing Apps*.

Approach

We use the number of different abstracted execution events as a measurement of the size of *CI*. We also study the *CI* changes by measuring the percentages of unchanged, added and deleted execution events. Given the current release *n* and the previous release *n-1*, the percentages of unchanged, added and deleted execution events are defined by the ratio of the number of unchanged, added or deleted events in release *n* over the number of total execution events in the previous release (*n-1*), respectively. For example, the percentage of unchanged execution events in release *n* ($P_{\ unchanged\ n}$) is calculated as:

$$P_{\ unchanged\ n} = \frac{\#\ unchanged\ events_n}{\#\ total\ events_{\ n-1}} \quad (1)$$

We identify modified events by manually examining added and deleted events. We use the frequency of execution events in both releases to assist in mapping between modified events with similar wording and frequencies across releases. Given the large number of events and releases in the *EA*, we only examine the top 40 most occurring events since they represent more than 90% of the logs. We use a similar equation as (1) for *EA*, except that the number of *total execution events* for *EA* is always 40.

Results

We first study the size of *CI* in the history of both systems. Figure 2 shows the growth trend of *CI* in *Hadoop*. We note that the *CI* in the last studied release (0.21.0) is 2.8 times the size of the first-studied release (0.14.0). In particular, the size of *CI* increases significantly in release 0.21.0 even though the size of the corresponding source code decreases by 20%. We also note that the *CI* increases more between major releases than between minor releases.

For the *EA* system, since we only study 2 major releases, we study the size of *CI* in each major release instead of generalizing a trend over the 2 studied releases. The *CI* size in the 2 major releases (9 releases in total) of *EA* is shown in Figure 3 (actual size is not shown due to NDA). For the *EA* system, we note that the size of *CI* does not change significantly in the first major release, while the *CI* increases significantly in the second major release. The *CI* of the last studied release (0.2.1) is 1.5 times the size of the first-studied release (0.1.0).

A closer analysis of the *CI* across releases shows that for both systems most (over 60%) of the old *CI* remains the same in new major releases (see Table IV and V). The *CI* events are more stable across minor releases (around 75-82% remains the same). This is good news for developers of *Log Processing Apps*. However, on average around 20% (for *EA*) and 31% (for *Hadoop*) of the *CI* are the same event but with different context. Such *CI* are troublesome for maintainers of *Log Processing Apps* since they might need to modify their code to account for such changes.

The large increase of *CI* size in major releases indicates that additional maintenance effort might be needed for *Log Processing Apps* to continue operating correctly even if the existing *Log Processing Apps* do not use *CI* about new features or the additional *CI* about the currently analyzed features.

It is important to note that all these *CI* changes are for a fixed set of executed features, i.e., although the features remain the same, the *CI* clearly does not. We study the release

TABLE IV
PERCENTAGE OF UNCHANGED, ADDED, DELETED AND MODIFIED *CI* IN THE HISTORY OF *Hadoop* (BOLD FONT INDICATES LARGE CHANGES).

Release	# Total	% Unchanged	% Added	% Deleted	% Modified
0.15.0	60	81.67	18.33	18.33	8.33
0.16.0	65	86.67	21.67	13.33	3.33
0.17.0	70	87.69	20.00	12.31	9.23
0.18.0	93	**51.43**	**81.43**	**48.57**	**28.57**
0.19.0	113	80.65	40.86	19.35	5.38
0.20.0	121	76.99	30.09	23.01	4.42
0.20.1	118	89.26	8.26	10.74	2.48
0.20.2	124	10000	5.08	0.00	0.00
0.21.0	168	**52.42**	**83.06**	**47.58**	**20.16**

TABLE V
PERCENTAGE OF UNCHANGED, ADDED, DELETED AND MODIFIED *CI* IN THE HISTORY OF *EA* (BOLD FONT INDICATES LARGE CHANGES).

Release	% Unchanged	% Added	% Deleted	% Modified
0.1.1	97.18	8.33	2.82	0.00
0.1.2	**63.05**	**34.27**	**36.95**	**42.50**
0.1.3	86.93	24.76	13.07	50.00
0.1.4	79.93	13.05	20.07	12.50
0.1.5	83.84	14.04	16.16	15.00
0.1.6	96.48	3.52	3.52	5.00
0.2.0	**78.21**	**49.53**	**21.79**	**17.50**
0.2.1	85.59	29.77	14.41	5.00

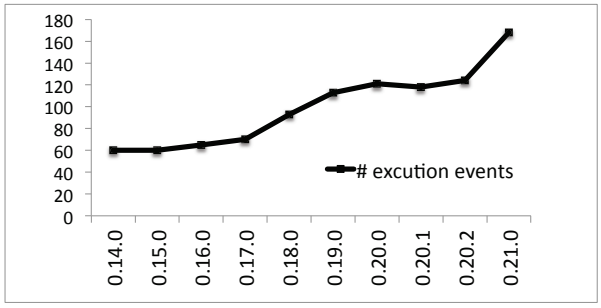

Fig. 2. Growth trend of *CI* in *Hadoop*.

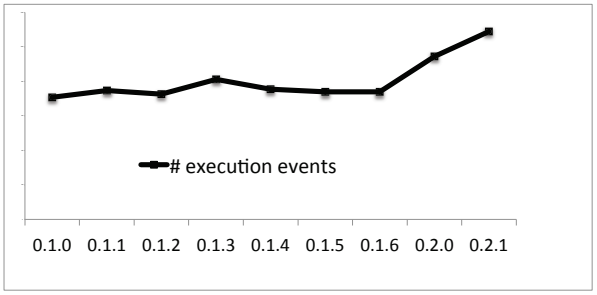

Fig. 3. Size of *CI* in 2 main releases and 7 minor releases of *EA*.

information for both releases and read through the change logs to better understand the rationale for large *CI* changes. We find that internal implementation changes often have a big impact on the *CI*. For example, according to Table IV, release 0.18.0 (in bold font) is one of the releases with the highest percentage of *CI* changes. Release 0.18.0 introduced new Java classes for *Hadoop* jobs (a core functionality of *Hadoop*) to replace the old classes. Release 0.21.0 officially replaced the old MapReduce implementation named "mapred", with a new implementation named "MapReduce". Similarly, the releases 0.1.3 and 0.2.0 (bold in Table V) of *EA* have significant behavioural and architectural changes compared with their previous releases.

It appears that the studied systems communicate a significant amount of implementation-level information, leading their *CI* to vary considerably due to internal changes.

> *The* CI *for both studied systems tends to contain implementation-level information. This provides* Log Processing Apps *with detailed knowledge of the internals of the systems. However, this makes such* Log Processing Apps *fragile, requiring continuous maintenance, in particular for major releases (around 40% of the* CI *changes). For minor releases, we still see a large percentage (20%-25%) of* CI *changes.*

RQ2: What types of modifications happen to *CI*?

Motivation

RQ1 shows that 20-30% of communicated events change with only their context information being modified (modified *CI*). These modified *CI* have a crucial impact on *Log Processing Apps*, since *Log Processing Apps* expect certain context information and are likely to fail when operating on events with modified context. In contrast, newly added *CI* is not likely to impact already developed *Log Processing Apps* since those applications are unaware of the new *CI*. In short, changes to the context of previously communicated events are more likely to introduce bugs and failures in *Log Processing Apps*. For example, during the history of *Hadoop*, "task" (an important concept of the platform) was renamed to "attempt", leading to failures of monitoring tools and to confusion within the user community about the communicated context [16]. Therefore, we wish to understand how communicated contexts change.

Approach

We follow a grounded theory [17] approach to identify modification patterns to the context. We manually study all events with a modified context. For each of them, we analyze what information is modified and how is the information modified. We repeat this process several times until a number of modification types emerge. We then calculate the distribution of different types of context modifications. The percentage is calculated as the ratio of the number of occurrence of a type in all the releases over the total number of modifications in all the releases. For example, the percentage of modified *CI* of type p ($P_{\ modified\ _p}$) is calculated as:

$$P_{\ modified\ _p} = \frac{\#\ modified\ events\ _p}{\#\ total\ modified\ events} \qquad (2)$$

Results

TABLE VII
PERCENTAGE OF AVOIDABLE, RECOVERABLE AND UNAVOIDABLE *CI* MODIFICATION IN *Hadoop* AND *EA*.

	Hadoop (%)	**EA (%)**
Avoidable	71.83	40.68
Recoverable	14.08	52.54
Unavoidable	14.09	6.78

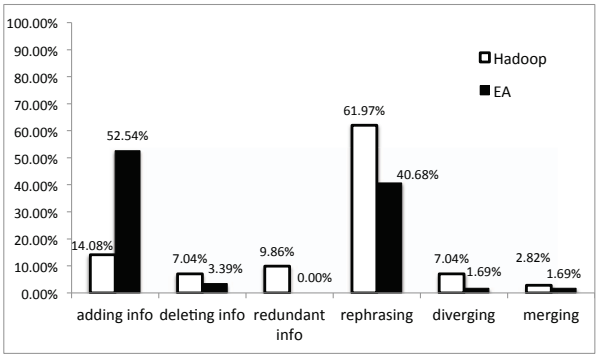

Fig. 4. Distributions of the different types of communicated context modifications for *Hadoop* and *EA*.

Table VI tabulates the six types of *CI* identified from our manual examination. The table defines each type and gives a real-life example of it from the studied data. Among all the types, *Rephrasing context* and *Redundant context* are avoidable modifications, since neither of them brings any additional information to the *CI*, and only cause avoidable changes in *Log Processing Apps*. The *Adding context* modification is typically unavoidable, but a robust log parser should still be able to parse the log correctly just by discarding the added information. Therefore, *Adding context* is a recoverable modification and has a less negative impact than the avoidable modifications. The other 3 types of modification, i.e., *Merging context*, *Splitting context* and *Deleting context*, are also unavoidable, but the *Log Processing Apps* might need to adapt for these modifications. Developers of the system should well document these modifications and inform people who make use of the modified *CI*.

Figure 4 shows the classification distribution of the *CI* modification types. Table VII shows the percentage of avoidable, recoverable and unavoidable *CI* modifications. We find that the majority of the *CI* modifications are either avoidable or recoverable. Only a small portion of the *CI* modifications is unavoidable. Simply put, developers can improve the maintenance of the *Log Processing Apps* by avoiding the avoidable modifications and documenting the unavoidable *CI* modifications.

Table VIII and IX show the detailed percentage of context modifications for both systems, broken down per version and pattern. Table VIII shows that the two largest kinds (in bold) of context modifications are both instances of *Rephrasing context*. They were introduced in release 0.18.0 and 0.21.0 of *Hadoop*. Table IX shows that many *Rephrasing context* instances are introduced in version 0.1.2 of *EA*. As noted in RQ1, all these three releases (0.18.0 and 0.21.0 of *Hadoop* and 0.1.2 of *EA*) have major changes. These results indicate that most of the *Rephrasing context* modifications may have a high correlation to the major changes introduced into the software systems. For example, in release 0.21.0, the old MapReduce library, which is the most essential part of *Hadoop*, was replaced by a whole new implementation. Therefore, the word "mapred" was replaced by the word "MapReduce". As both implementations have the same features, the operator should not need to worry about such implementation changes of the library. However, such *Rephrasing context* modifications require updating any *Log Processing Apps* to ensure their proper operation.

In contrast, even though release 0.1.3 of the *EA* has many *Adding context* modifications, it does not have a large number of added or deleted *CI*. This result indicates that even though some releases do not introduce major changes into the system, *CI* may still be modified significantly.

*We have identified six types of communicated context modifications. Two of the types (*Rephrasing context *and* Redundant context*) are avoidable, one type (*Adding context*) is unavoidable but its impact can be controlled through the use of robust parsing techniques. The other three types (*Merging context*,* Splitting context *and* Deleting context*) are also unavoidable and have a high chance to introduce errors. Around 85-93% of the modifications can be controlled through careful attention by system developers (avoidable context modifications) or careful robust programming of* Log Processing Apps *(*Adding Context*).*

RQ3: What information is conveyed in short-lived *CI*?

Motivation

RQ1 shows that there is added and deleted *CI* in every release. Some *CI* is added by developers and removed in a short period of time. The *Log Processing Apps* depending on such short-lived *CI* may be fragile. We study the information conveyed in the short-lived *CI* to understand why such *CI* exists only within a short period of time. By studying the conveyed information, we can understand the *CI* at a high level of abstraction instead of simple counts of added, removed and modified *CI* like in the previous two questions.

Approach

We consider communicated events that only exist in a single release as *short-lived CI*. To understand the information conveyed by the short-lived *CI* over time, we generate a Latent Dirichlet Allocation (LDA) [18] model. We put the short-lived *CI* of each release in a separate file as input documents of LDA. We use *MALLET* [19] to generate the LDA models with 5 topics. Each word in the topics has a probability indicating the significance of such word in the corresponding topic. We generated the 5 words with the highest probability in each topic to determine the information conveyed by *CI* in the topic. Finally, we examine the words in the 5 topics and generalize a

TABLE VI
CI MODIFICATION TYPES AND EXAMPLES.

Pattern	Definitions	Examples	
		Before	After
Adding context	Additional context is added into the communicated information.	ShuffleRamManager memory limit *n* MaxSingleShuffleLimit m	ShuffleRamManager memory limit *n* MaxSingleShuffleLimit *m* mergeThreshold *Q*
Deleting context	Context is removed from the communicated information.	Got *n* map output known output *m*	Got *n* output
Redundant context	Some redundant information is added into the context or the added information can be generated without being included in the context.	Start task tracker at machine *A*	Start task tracker at machine *A* IP *X*
Rephrasing context	The context is rephrased to the new context.	Hadoop mapred Reduce task fetch *n* bytes	Hadoop MapReduce task Reduce fetch *n* bytes
Merging context	Several old contexts are merged into one log event.	MapTask data buffer MapTask data buffer	MapTask buffer
Splitting context	The old context is split into multiple new contexts.	Adding task to tasktracker	Adding Map Task to tasktracker; Adding Reduce Task to tasktracker

TABLE VIII
DETAILED PERCENTAGES OF DIFFERENT TYPES OF CONTEXT MODIFICATIONS IN *Hadoop*.

Release	Adding context	Deleting context	Redundant context	Rephrasing context	Merging context	Splitting context
0.15.0	1.41	0.00	2.82	2.82	0.00	0.00
0.16.0	0.00	0.00	2.82	0.00	0.00	0.00
0.17.0	0.00	0.00	0.00	8.45	0.00	0.00
0.18.0	0.00	0.00	0.00	**28.17**	0.00	0.00
0.19.0	0.00	2.82	0.00	4.23	0.00	0.00
0.20.0	2.82	1.41	1.41	0.00	1.41	0.00
0.20.1	0.00	1.41	0.00	1.41	1.41	0.00
0.20.2	0.00	0.00	0.00	0.00	0.00	0.00
0.21.0	9.86	1.41	1.41	**16.9**	2.82	2.82

TABLE IX
DETAILED PERCENTAGES OF DIFFERENT TYPES OF CONTEXT MODIFICATIONS IN *EA*.

Release	Adding context	Deleting context	Redundant context	Rephrasing context	Merging context	Splitting context
0.1.1	0.00	0.00	0.00	0.00	0.00	0.00
0.1.2	6.78	1.69	0.00	**20.34**	0.00	0.00
0.1.3	**22.03**	0.00	0.00	10.17	0.00	1.69
0.1.4	6.78	0.00	0.00	1.69	0.00	0.00
0.1.5	8.47	1.69	0.00	0.00	0.00	0.00
0.1.6	1.69	0.00	0.00	1.69	0.00	0.00
0.2.0	6.78	0.00	0.00	3.39	1.69	0.00
0.2.1	0.00	0.00	0.00	3.39	0.00	0.00

one-sentence summary based on our knowledge to understand the information conveyed in short-lived *CI*.

Results

A manual analysis of short-lived *CI* reveals that some *CI* corresponds to exceptions and stack traces. We removed such data since it does not represent short-lived *CI* but primarily rare errors.

Table X shows the topics generated by LDA. The words in each topic are sorted by their degree of memberships. From the results in Table X, we find that the topics in short-lived *CI* contain implementation-level information. For example, the word "ipc" in topic 4 means inter-procedural communication between machines, which is an implementation detail of *Hadoop*. In addition, the information about outputting results and choosing a server in topics 1, 2 and 5 is also implementation-level information. We performed the same study on *EA*, with the results similar to the results of *Hadoop*.

Some *Log Processing Apps* are designed for recovering high-level information about the systems, e.g., system workload rather than implementation details. Such *Log Processing Apps* would not need the implementation-level information. However, there are a few *Log Processing Apps* that are designed for debugging purposes. Such applications require

TABLE X
TOPICS OF THE SHORT-LIVED *CI* IN *Hadoop* GENERATED BY LDA.

#	Topic	Summary
1	job output node jobhistory saved	Hadoop saves output to a machine.
2	reducetask jobinprogress choosing server hadoop	Hadoop assigns a reduce task to a machine.
3	mapred map tracker taskinprogress jobtracker	Map task updates its progress.
4	id org local file ipc	Hadoop reads from a local file.
5	task tasktracker attempt outputs tip	Hadoop Attempt saves its output and reports to the task tracker.

the implementation-level information in the short-lived *CI*, and would be fragile as their corresponding *CI* is continuously changing.

Short-lived CI *contains implementation-level details to facilitate system development. Log Processing Apps depending on implementation-level information are likely to be more fragile. More maintenance efforts are needed for* Log Processing Apps *that depend on implementation-level information.*

V. THREATS TO VALIDITY

This section presents the threats to validity of our study.

External validity

Our study is an exploratory study performed on *Hadoop* and an enterprise application. Even though both systems have years of history and large user bases, more case studies on other software systems in the same domain are needed to see whether our findings can generalize. Similarly, the studied logs are collected from specific workloads, which may not generalize. We plan to study in-field execution logs in our future work.

Internal validity

Our study includes several manual steps, such as the analysis of log modifications and the classification of log reformatting. Our findings may contain subjective bias.

Our study is performed on both major and minor releases of *Hadoop* and *EA*. However, the major and minor releases in the two systems may not contain similar amount of source code changes. We study *Hadoop* with mostly major releases while we study *EA* with mostly minor releases. The major releases of *Hadoop* may not contain as significant changes and the minor releases of *EA* may contain large numbers of changes. Therefore, our findings about major and minor releases may be biased. We plan to study more systems in the same fields to counter this bias.

Construct validity

We use execution logs to study the communicated information. Other types of *CI*, such as code comments, may not evolve in the same manner or contain the same information as execution logs. Studying other types of *CI* is part of our future work.

Our study only focus the information conveyed by *CI*, but does not have any knowledge of the actual source code related to the *CI*. For example, a log line is output before a system event in the first release but after the system event in the second release. We cannot detect such changes by merely studying the execution logs. We plan to combine the analysis of source code and execution logs in our future work to overcome this limitation.

Our study is mainly based on the abstraction of execution events proposed by Jiang *et al*. [14]. This approach, customized to better fit the two subject systems, is shown to have high precision and recall. However, falsely abstracted log events may still exist, which may potentially bias our results. We plan to adopt other log abstraction techniques to improve the precision and to reduce the falsely abstracted execution events in our study.

VI. DISCUSSION AND RELATED WORK

In this section, we discuss the related topics and prior work related to our study.

A. Non-code based evolution studies

While many prior studies examined the evolution of source code, (e.g., [20]–[22]), this paper studies the evolution of software systems from the perspective of non-code artifacts associated with these systems. The non-code artifacts are extensively leveraged in software engineering practice, yet the dependency between such artifacts and their surrounding ecosystem lacks tracking. Therefore, understanding the evolution of non-code based software artifacts is important. For example, the evolution of the following non-code artifacts has been studied before:

- **System Documentation:** Software systems evolve throughout the history, as new features are added and existing features are modified due to bug fixes, performance and usability enhancements. Antón *et al*. [23] study the evolution of telephony software systems by studying the user documentation of telephony features in the phone books of Atlanta.
- **User Interface:** His *et al*. [24] study the evolution of Microsoft Word by looking at changes to its menu structure. Hou *et al*. [25] study the evolution of UI features in the Eclipse IDE.
- **Features:** Instead of studying the code directly, some studies have picked specific features and followed their implementation throughout the lifetime of the software system. For example, Kothari *et al*. [26] propose a technique to evaluate the efficiency of software feature development by studying the evolution of call graphs generated during the execution of these features. Our study is similar to this work, except for using *CI* instead

of call graphs. Greevy *et al*. [27] use program slicing to study the evolution of features.

- **Communicated Information:** We divide *CI* into *CI* about code (static *CI*) and *CI* about the execution (dynamic *CI*).
  - **Evolution of Static *CI*:** A major examples of this *CI* are code comments, which are a valuable instrument to preserve design decisions and to communicate the intent of the code to programmers and maintainers. Jiang *et al*. [28] study the evolution of source code comments and discover that the percentage of functions with header and non-header comments remains consistent throughout the evolution. Fluri *et al*. [29], [30] study the evolution of code comments in 8 software projects.
  - **Evolution of Dynamic *CI*:** To the best of our knowledge, this paper is the first work that seeks to study the evolution of dynamic *CI*.

B. Logs as a source of CI

Our case studies use execution logs as a primary source of *CI*. This is based on our team's extensive experience working with several large enterprises. While many systems today support monitoring APIs to communicate to a system, all too often users of such systems still make extensive use of execution logs as a valuable source of communicated information about the execution of these systems.

In many ways, the logs provide a non-typed, flexible communication interface to the outside world. The flexible nature of the logs makes them easy to evolve by developers (hence faster to respond to changes in the systems). However, this also makes the applications that depend on them very fragile. As additional applications depend more and more on specific *CI*, it is often the case that such information is then formalized and communicated through more formalized and typed interfaces. For example, ARM [31] (Application Response Measurement) provides monitoring APIs to assist in system response time monitoring and performance problem diagnosis. Further studies are needed to better understand the evolution of *CI* from very flexible to well defined APIs.

We coined the term *CI* to clearly differentiate it from tracing information. Tracing information is mainly low-level and generated in a blind way, whereas *CI* is intentionally generated by software developers who consider the value of using such information in practice. We firmly believe that *CI* can (and should) remain consistent even as the lower-level implementation details of an application change. Recent work by Yuan *et al*. [32], has explored how one can improve the *CI* to easily detect and repair bugs by enhancing the communicated contexts. In future studies, we wish to define metrics to measure the quality of *CI* and the need for *CI* changes relative to code changes.

C. Traceability between Logs and Log Processing Apps

Most software developers consider logs as a final output of their systems. In reality, logs are just the input for a whole range of applications that live in the log-processing ecosystem surrounding these systems.

Our study is the first study to explore how changes in parts of an ecosystem (communicated information, i.e., logs), once released in the field, might impact other parts of the system (*Log Processing Apps*). The need for such types of studies was noted by Godfrey and German [33], as they recognized that most software systems today are linked in a formal or informal manner with other systems within their eco-systems.

Lehman's earlier work [22] recognizes the need for applications to adapt to the changes in their surrounding environment. In this study, we primarily focused on the environmental changes (i.e., changes to *CI*). In future work, we wish to study the changes in all aspects of the eco-system, namely the system, the *CI*, and the *Log Processing Apps* that process the *CI*.

Our study and our industrial experience support us in advocating the need for research on tools and techniques to establish and maintain traceability between the *CI* (logs in our case) and the *Log Processing Apps*. Such a line of work might increase the cost of building large systems. However, it is essential for reducing the maintenance overhead and costs for all apps within the eco-system of the system.

VII. Conclusion

Communicated information, such as execution logs and system events, is generated by snippets of code inserted explicitly by domain experts to record valuable information. An eco-system of *Log Processing Apps* analyzes such valuable information to assist in software testing, system monitoring and program comprehension. Yet, these *Log Processing Apps* highly depend on *CI* and are hence impacted by changes to the *CI*. In this paper, we have performed an exploratory study on the *CI* of 10 releases of an open source software named *Hadoop* and 9 releases of a legacy enterprise application.

Our study shows that systems communicate more about their execution as they evolve. During the evolution of software systems, the *CI* also evolves. Especially when there are major source code changes (e.g., a new major release), the *CI* is changed significantly, although the changes of implementation ideally should not have an impact on *CI*. In addition, we observed 6 types of *CI* modifications. Among the *CI* modifications in the studied systems, only less than 15% of the modifications are unavoidable and are likely to introduce errors into *Log Processing Apps*. We also find that short-lived *CI* typically contains system implementation-level details.

Our results indicate that additional maintenance resources should be allocated to maintain *Log Processing Apps*, especially when major changes are introduced into the software systems. Because of the evolution of *CI*, traceability techniques are needed to establish and track the dependencies between *CI* and the *Log Processing Apps*.

However, even today, without traceability techniques between *CI* and *Log Processing Apps*, the negative impact can still be minimized by both the system developers (who generate *CI*) and the developers of *Log Processing Apps* (who

consume *CI*). System developers should avoid modifying *CI* as much as possible. The avoidable *CI* modifications include rephrasing and adding redundant information in the *CI*. On the other hand, *Log Processing App* developers should write robust log parsers to avoid the impact of *CI* changes. In addition, more resources should be allocated to maintain *Log Processing Apps* designed for debugging problems (from short-lived *CI*).

ACKNOWLEDGMENT

We are grateful to Research In Motion (RIM) for providing access to the enterprise application used in our case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of RIM and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of RIM's products.

REFERENCES

[1] B. Cornelissen, A. Zaidman, A. V. Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Engineering*, vol. 35, pp. 684–702, 2009.

[2] 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*. Leipzig, Germany: ACM, 2008, pp. 713–723.

[3] M. Nagappan, K. Wu, and M. A. Vouk, "Efficiently extracting operational profiles from execution logs using suffix arrays," in *ISSRE'09: Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*. Bengaluru-Mysuru, India: IEEE Press, 2009, pp. 41–50.

[4] 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*. Vancouver, BC, Canada: USENIX Association, 2010, pp. 7–7.

[5] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *ICSM '08: Proceedings of 24th IEEE International Conference on Software Maintenance*. Beijing, China: IEEE, 2008, pp. 307–316.

[6] ——, "Automated performance analysis of load tests," in *ICSM '09: 25th IEEE International Conference on Software Maintenance*. Edmonton, Alberta, Canada: IEEE, 2009, pp. 125–134.

[7] "Infosphere streams," http://goo.gl/nI1a4.

[8] "Splunk," http://www.splunk.com/.

[9] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, January 2008.

[10] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang, "Chukwa, a large-scale monitoring system," in *CCA '08: Proceedings of the first workshop on Cloud Computing and its Applications*, Chicago, IL, 2008, pp. 1–5.

[11] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Salsa: analyzing logs as state machines," in *WASL'08: Proceedings of the First USENIX conference on Analysis of system logs*. San Diego, California: USENIX Association, 2008, pp. 6–6.

[12] B. Beizer, *Software system testing and quality assurance*. New York, NY, USA: Van Nostrand Reinhold Co., 1984.

[13] 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: Proceedings of the 6th international conference on Aspect-oriented software development*. Vancouver, British Columbia, Canada: ACM, 2007, pp. 199–211.

[14] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *J. Softw. Maint. Evol.*, vol. 20, no. 4, pp. 249–267, 2008.

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

[16] "Hadoop 0.18.0 release notes," http://goo.gl/zW1qa.

[17] R. Brower and H. Jeong, "Beyond description to derive theory from qualitative data," in *Handbook of Research Methods in Public Administration*, B. Raton, Ed. Taylor Francis, 2008, pp. 823–839.

[18] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, March 2003.

[19] "Mallet: A machine learning for language toolkit," http://http://mallet.cs.umass.edu/.

[20] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth, "Software Evolution Observations Based on Product Release History," in *ICSM '97: Proceedings of the International Conference on Software Maintenance*. Bari, Italy: IEEE Computer Society, 1997, pp. 160–166.

[21] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in *ICSM '00: Proceedings of the International Conference on Software Maintenance*. San Jose, California, USA: IEEE Computer Society, 2000, pp. 131–142.

[22] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and Laws of Software Evolution - The Nineties View," in *Proceedings of the 4th International Symposium on Software Metrics*. Albuquerque, NM, USA: IEEE Computer Society, 1997, pp. 20–32.

[23] A. I. Antón and C. Potts, "Functional paleontology: system evolution as the user sees it," in *Proceedings of the 23rd International Conference on Software Engineering*. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 421–430.

[24] I. His and C. Potts, "Studying the Evolution and Enhancement of Software Features," in *ICSM '00:Proceedings of the International Conference on Software Maintenance*. San Jose, California, USA: IEEE Computer Society, 2000, pp. 143–151.

[25] D. Hou and Y. Wang, "An empirical analysis of the evolution of user-visible features in an integrated development environment," in *CASCON '09:Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. Toronto, Ontario, Canada: ACM, 2009, pp. 122–135.

[26] J. Kothari, D. Bespalov, S. Mancoridis, and A. Shokoufandeh, "On evaluating the efficiency of software feature development using algebraic manifolds," in *ICSM '08: International Conference on Software Maintenance*, Beijing, China, 2008, pp. 7–16.

[27] O. Greevy, S. Ducasse, and T. Gîrba, "Analyzing software evolution through feature views: Research Articles," *J. Softw. Maint. Evol.*, vol. 18, pp. 425–456, November 2006.

[28] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in postgresql," in *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*. Shanghai, China: ACM, 2006, pp. 179–180.

[29] B. Fluri, M. Wursch, and H. C. Gall, "Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes," in *WCRE '07:Proceedings of the 14th Working Conference on Reverse Engineering*. Vancouver, BC, Canada: IEEE Computer Society, 2007, pp. 70–79.

[30] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Control*, vol. 17, pp. 367–394, December 2009.

[31] M. W. Johnson, "Monitoring and Diagnosing Application Response Time with ARM," in *Proceedings of the IEEE Third International Workshop on Systems Management*. Newport, RI, USA: IEEE Computer Society, 1998, pp. 4–15.

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

[33] M. W. Godfrey and D. M. Germán;, "The past, present, and future of software evolution," in *FoSM: Frontiers of Software Maintenance*, Beijing, China, October 2008, pp. 129–138.