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
Email: {swy, zmjiang, bram, ahmed}@cs.queensu.ca

Michael W. Godfrey
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
Email: 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. In this paper, we turn our attention to another source of information, i.e., the communicated information about the execution of a software system. Execution logs and system events are an example of communicated information. The logs are generated from statements that are inserted on purpose by domain experts (e.g., developers or operators) to signify points of interest instead of automatically instrumented after the fact, without considering domain knowledge. The accessibility and domain-driven nature of the communicated information make it an ideal source to study the evolution of a software system. Through a case study on one large open source and another industrial software system, we explore the concept of communicated information and its evolution by mining the execution logs of these systems. Our study shows that over time systems are willing to communicate more about themselves and that the communicated information changes over time. We also find that some of the communicated information is short-lived while other information is long-lived. The short-lived information corresponds to developer-level information (e.g., opening a database connection), while the longer-lived information corresponds to domain-level information (e.g., opening a new user account).**

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

I. Introduction

Software engineering researchers traditionally use software profiling and instrumentation techniques to generate execution traces for studying the run-time behaviour of software systems [1]. However, such techniques impose high overhead and slow down the execution, especially for real-life workloads. In addition, software profiling and instrumentation is done after the fact without domain knowledge of the software system. Therefore, massive instrumentation often leads to hardly interpretable results.

In practice, system administrators and developers typically rely on the Communicated Information (*CI*), which consists fundamentally of system events, of ultra-large software (ULS) systems to understand field behaviour of large systems and to diagnose and repair bugs. Execution log is one way to persistently store *CI*. Such logs typically contain information about the system activities (events) and their associated contexts. The code to generate logs is explicitly added to the code by domain experts to communicate important information about the execution of a system. The importance of the information varies based on the purpose of the logs. For example, detailed debugging logs are relevant to developers, while operation logs summarizing the key execution steps are more relevant to operators and administrators.

The rich nature of the *CI* has introduced a whole new market of applications that complement ULS systems. We collectively call these applications *Log Processing Apps*. *Log Processing Apps* are, for example, used to generate workload information for large-scale systems capacity planning [2], [3], to monitor system health [4], to detect system abnormal behaviours [5], or to flag performance degradations [6]. Often such applications are in-house applications requiring continuous evolution as the needs change and as the *CI* changes. Companies, such as IBM [7] and Splunk [8], support the development of such applications by providing high-level domain specific frameworks or visual development environments. 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 legacy enterprise application, which we name *EA*. Our study is the first step in understanding the maintenance requirement for the field of *Log Processing Apps* by studying the evolution of the *CI*. This paper addresses the following research questions.

RQ1: How does the *CI* change over time?
We find that the size of the *CI* keeps on increasing, but over all, majority of the *CI* remains constant. On average, 69.2% of the *CI* in *Hadoop* and 78.9% in *EA* do not change from release to release.

RQ2: What types of modifications happen to *CI*?
We identify six reasons for modifying the context associated with *CI*. The rephrasing of the context has the most harmful impact on the *Log Processing Apps*, since it leads to changes to log lines without change to the *CI*, causing unnecessary changes to the *Log Processing Apps*.

RQ3: What information is conveyed in the *CI*?
We find that the long-lived CI mainly focuses on the high-level (e.g., domain level) concepts of the software systems. The short-lived CI mainly talks about implementation-level details.

Our study highlights the importance of *CI* and the fact that tools and techniques are needed to ease the maintenance of *Log Processing Apps*. Such techniques should establish and maintain traceability between *CI* and the *Log Processing Apps* that make use of them.

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

II. A Motivating Example

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

The example is about 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 1

System operators identified a performance problem in the prior release. In order to diagnose the problem, developer A added more information to the execution event in *CI*, such as the ID of the user who started uploading the file. Using the added context in the execution logs, the system operators resolved the performance 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 2

The file upload feature was overhauled, leading the developers to change the communicated events and their associated logs. The context changes to the log files led to failures of the *Log Processing Apps*. The application started warning of a performance problem. After several hours of analysis, the root of the false error was identified. The *CI* had been changed by a developer who was not aware (no traceability) that others made use of this information.

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

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*.

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

TABLE I
OVERVIEW OF THE STUDIED RELEASES OF *Hadoop*

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

III. Case Study Setup

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

A. Studied systems

We choose one open source software 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 an open source software application named *Hadoop*[1], and 9 release of a closed source large enterprise application (*EA*). To improve readability, we mark minor releases in italic forms.

Hadoop is a large distributed data processing platform that implements the MapReduce [9] data processing paradigm. We pick 10 releases of *Hadoop*, shown in Table I, from 0.14.0 to 0.21.0, as 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 the studied releases, 0.20.1 and 0.20.2 (in italic) are minor releases in the 0.20.0 series.

The enterprise application (*EA*) in our study is a large-scale, legacy software application. Due to a Non-Disclosure Agreement, we cannot reveal additional details about the enterprise application. We do note that it is considerably larger

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

than *Hadoop* and with a very large user base. We choose 9 releases of *EA* in our study, including respectively 7 and 2 minor releases from two consecutive major releases. 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.

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 sought to understand the *CI* of each system based on the execution of a consistent set of features for the various 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 large enterprise application running in the field.

Data collection

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

Hadoop workload

The *Hadoop* workload consists of two example applications, *wordcount* and *grep*. The *wordcount* application generates the frequencies of all the words in the input data and the *grep* application searches the input data for a certain 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* application in our study is a basic string ("fail").

The Enterprise System workload

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 ensures that all commonly used features are exercised.

We perform the same 8-hour standard load test [10] on each of the releases of *EA*. A standard load test mimics the real-life usage of the application and also ensures that all of the application features are covered during the test, such that the collected execution logs are consistent over different runs of *EA*.

Log abstraction

We recover the *CI* by mining the generated logs. However, execution logs (e.g., Table II) typically do not follow strict formats. Instead, they are often inconsistent [11]. 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 describe the system events (execution events), while the dynamic values indicate the execution *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 the technique proposed by Jiang *et al*. [12] to automatically extract the execution events and their associated context. The precision and recall of this technique are both over 80%. 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., 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 execution events as representations of communicated system events.

TABLE II
EXAMPLE OF EXECUTION LOG LINES

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

TABLE III
ABSTRACTED EXECUTION EVENTS

Event	Event template	#
E1	time=$t, Task=Trying to launch, TaskID=$id	1,2
E2	time=$t, Task=JVM, TaskID=$id	3,5
E3	time=$t, Task=Reduce, TaskID=$id	4,6,7
E4	time=$t, Task=Progress, TaskID=$id	8
E5	time=$t, Task=Commit Pending, TaskID=$id	10
E6	time=$t, Task=Done, TaskID=$id	9,11

IV. CASE STUDY RESULT

In this section, we present the answers to our research questions. For each research question, we present the motivation behind the question, our approach, and the corresponding results.

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

Motivation

The growth of the *CI* impacts the maintenance of *Log Processing Apps*. An overall growth of communicated system events would be an indication of the complexity of designing and maintaining *Log Processing Apps*. Continuous and frequent changes in the *CI* would indicate extra maintenance costs for *Log Processing Apps*, since these applications typically must change to cope with each change to the *CI*. The frequent change of the *CI* makes *Log Processing Apps* fragile due to the lack of uniform traceability techniques between *Log Processing Apps* and *CI*.

Approach

In our study, we use the number of different abstracted execution events as a measurement of the amount of *CI*. In addition, we track the *CI* changes by measuring the percentages of unchanged, added and deleted abstracted execution events. Given the current release *n* and the previous release *n-1*, the percentages of unchanged, added and deleted execution activities 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:

$$P_{\ unchanged\ n} = \frac{\#\ unchanged\ events_n}{\#\ total\ events\ _{n-1}} \tag{1}$$

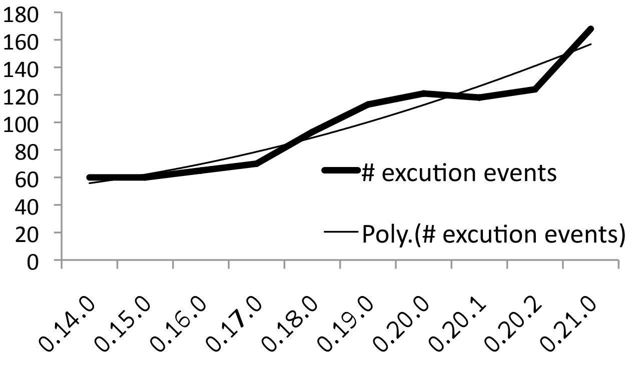
We manually examine all the added and the deleted events to identify modified events. To accurately identify the modified events, for each added and deleted execution events, we use the frequency of appearances of the events in both releases to assist in mapping between new events with similar wording and similar frequency of occurrences with the deleted events from previous releases. Therefore, the uncovered modified events are sub-set of added and deleted events. Given the large number of events and releases in the *EA*, we manually examine only top 40 most occurring events since they represent more than 90% of the data. We use similar equation as (1) for *EA*, except that the number of total execution events for *EA* is always 40.

Results

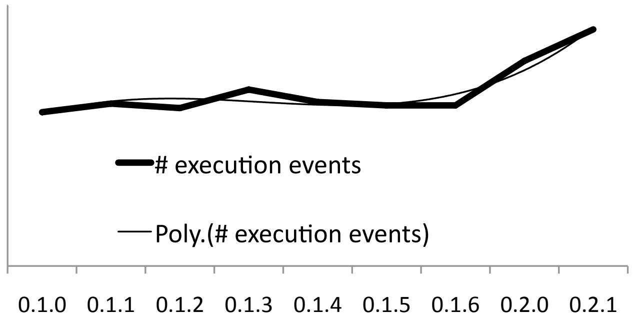
The size of *CI* is growing super-linearly for both systems. Figure 1 shows the growth trend of *CI* in both studied systems on a linear scale. For the *EA* system, we do not show the values on the Y axis. For *Hadoop*, we note that the *CI* in the last studied release (0.21.0) is 2.8 times the size of the *CI* in the first studied release (0.14.0). We also note that *CI* increases more between major releases than between minor releases. For the *EA* system, we note a stable trend in the first seven releases, while the *CI* increases significantly in the last two releases (new major release).

Table IV and Table V show the percentage of added, deleted and modified execution events. On average, 69.2% of the *CI* for *Hadoop* remains the same from release to release. In the added or deleted *CI*, 30.7% of it is modified, i.e., the same communicated events occur, but with different contexts attached to them. For the *EA* system, 78.9% of the *CI* stays the same. Among the rest of the examined top 40 events, on average 19.9% of them are modified.

We observe that major changes to the software systems can lead to major changes in *CI*. For example, the releases 0.18.0 and 0.21.0 of *Hadoop* have the lowest percentages of unchanged *CI* (in bold), as well as a high percentage of added, deleted and modified execution events (in bold). We studied the release information for both releases and read through the change logs, to better understand the rationale for such large changes in the *CI*. 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) *Hadoop*



(b) *EA*

Fig. 1. Growth trend of *CI* in *Hadoop* (a) and *EA* (b).

a new implementation named "MapReduce". Similarly, the releases 0.1.3 and 0.2.0 of *EA* saw significant behavioural and architectural changes compared with their previous releases.

Since *Log Processing Apps* highly depend on the CI, our findings suggest that additional maintenance effort should be expected to maintain *Log Processing Apps* when major changes are introduced into their associated software systems.

Communicated Information exhibits an overall increasing trend. Almost 30% of the CI *changes on average across releases, which is an indication that* Log Processing Apps *require continuous maintenance once over time. Most of the major changes occur between major releases.*

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

Motivation

Our results in RQ1 show that a number of communicated events change with only their context information being modified (modified *CI*). These modified *CI* have a crucial impact on *Log Processing Apps* in the field, 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* are not likely to impact already developed *Log Processing Apps*. They will just be ignored. In short, changes to the context of previously communicated events are

TABLE IV
PERCENTAGE AND NUMBER OF UNCHANGED, ADDED, DELETED AND MODIFIED *CI* IN THE HISTORY OF *Hadoop*.

Release	# Total	%(#) Unchanged	%(#) Added	%(#) Deleted	%(#) Modified
0.15.0	60	81.67(49)	18.33(11)	18.33(11)	8.33(5)
0.16.0	65	86.67(52)	21.67(13)	13.33(8)	3.33(2)
0.17.0	70	87.69(57)	20.00(13)	12.31(8)	9.23(6)
0.18.0	93	**51.43(36)**	**81.43(57)**	**48.57(34)**	**28.57(20)**
0.19.0	113	80.65(75)	40.86(38)	19.35(18)	5.38(5)
0.20.0	121	76.99(87)	30.09(34)	23.01(26)	4.42(5)
0.20.1	118	89.26(108)	8.26(9)	10.74(12)	2.48(3)
0.20.2	124	99.15(117)	5.08(6)	0.00(0)	0.00(0)
0.21.0	168	**51.61(64)**	**83.06(103)**	**47.58(59)**	**20.16(25)**

TABLE V
PERCENTAGE OF UNCHANGED, ADDED, DELETED AND MODIFIED *CI* IN THE HISTORY OF *EA*.

Release	% Unchanged	% Added	% Deleted	% Modified
0.1.1	96.89	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

likely to introduce bugs and failures in *Log Processing Apps*. For example, during the history of *Hadoop*, the "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 [13]. Therefore, we wish to understand the different patterns of changes to communicated contexts.

Approach

We follow a grounded theory [14] approach to identify modification patterns to the context. We manually examine all events with modified context and associate patterns to their changes. We repeat this process several times until a number of change patterns emerge. We then calculate the distribution of different patterns of context modifications. The percentage is calculated as the ratio of the occurrence of a pattern in each release over the total number of modifications in all releases. For example, the percentage of modified *CI* in pattern p ($P_{\ modified\ p}$) is calculated as:

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

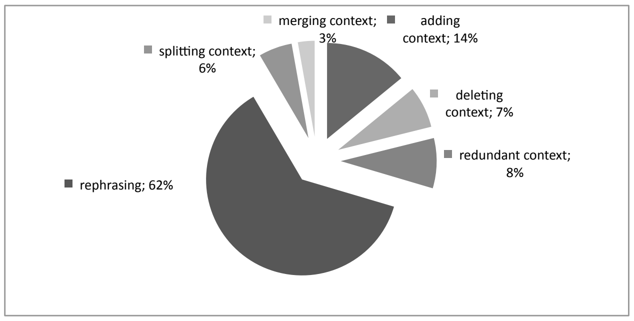
Results

Table VI tabulates the six identified patterns. The table defines each pattern and gives a real-life example of it. For example, the modification pattern *Rephrasing context* corresponds to the rewording of *CI* without any changes to the actual information.

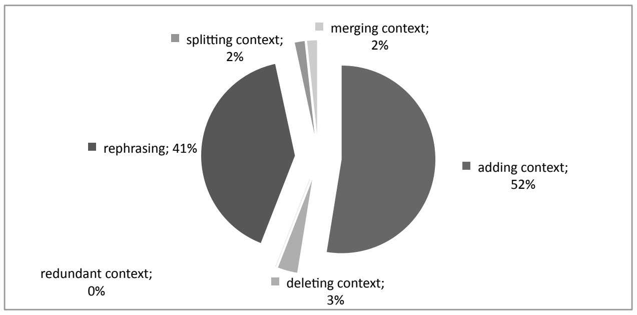
In the example of *Rephrasing context* shown in Table VI. The "mapred" word, short term for "MapReduce", is replaced by "MapReduce", and the word "Reduce task" is replaced by the word "task Reduce". Such modifications likely have no impact on human reading, but a *Log Processing App* that looks for the word "mapred" may generate incorrect results. *Rephrasing Context* is unnecessary and harmful execution event modification, which should be avoided entirely. Among all the other patterns, *Redundant context*, *Merging context* and *Splitting context* are also avoidable, since they do not change actual information in the *CI*. Other than *Rephrasing context*, *Deleting context* is also likely to have negative impact on the *Log Processing Apps*, since the applications may depend on the deleted *CI*.

Figure 2 shows that *Rephrasing context* and *Adding context* are two frequently occurring patterns. Over 60% of the context modifications in *Hadoop* are *Rephrasing context* pattern. For the *EA*, *Rephrasing context* represents 41% and *Adding context* represents 52% of the modifications. In short, over half of the evolution of *CI* in *Hadoop* and *EA* are likely to lead to harmful, yet unnecessary effects to the *Log Processing Apps*. Developers of large software systems should try to avoid modifying CI as much as possible.

Table VII and VIII show the detailed percentage of context modifications for both systems, broken down per version and pattern. Table VII shows that the two largest percentage (in bold) of contextual modifications are both instances of *Rephrasing context*. They were introduced in release 0.18.0 and 0.21.0 of *Hadoop*. Table VIII shows that a large number of *Rephrasing context* instances was introduced in 0.1.2 of *EA*. As noted in RQ1, all three releases have major changes introduced into the system. These results indicate that most of the *Rephrasing context* modifications may have correlation to the major changes introduced into the software systems. For example, in release 0.21.0, the old MapReduce library,

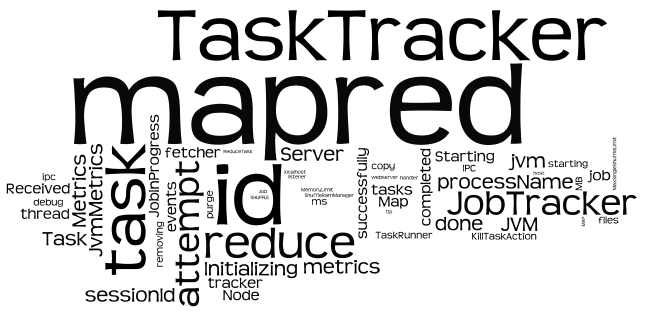


(a) *Hadoop*

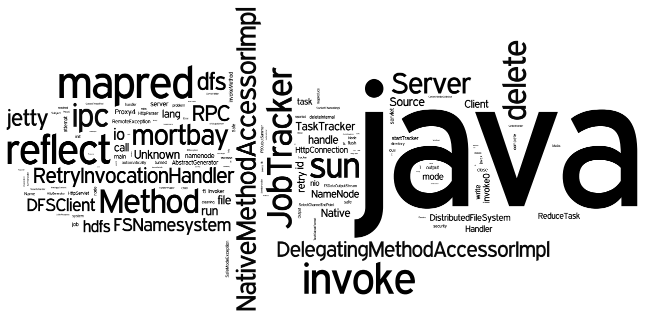


(b) *EA*

Fig. 2. Classification distributions of communicated context modifications in *Hadoop* (a) and *EA* (b).



(a) long-lived *CI*



(b) short-lived *CI*

Fig. 3. Tag clouds of top 50 words in long-lived (a) and short-lived (b) *CI* of*Hadoop*.

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 system operator should not worry about the changes to implementation details of the library. However, such *Rephrasing context* requires updating *Log Processing Apps* to ensure its proper operation.

In contrast, even though release 0.1.3 of the *EA* has a large number of *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.

Due to the harmful impact of *Rephrasing context* modifications, we recommend the allocation of additional maintenance resources to *Log Processing Apps* when major changes are introduced into the corresponding software system.

Six patterns of communicated context modifications are observed. Rephrasing context *and* Adding context *are the most frequently occurring of the identified modification patterns. The* Rephrasing context *pattern has the largest negative impact on* Log Processing Apps.

RQ3: What information is conveyed in the *CI*?

Motivation

By understanding the type of conveyed information, we can describe the evolution of *CI* at a high level of abstraction instead of simple counts of added, removed and modified *CI* as done in the previous two questions.

Approach

We study the information conveyed in long-lived and short-lived *CI* separately. We consider communicated events that exist across all releases as *long-lived CI*, while communicated events that only exist in a single release are as *short-lived CI*. To determine the most frequently used words in the execution events, we visualize the word tokens of the events using a tag cloud [15]. A tag cloud adjusts the size of each word according to its frequency of occurrence relative to all the words. More frequently mentioned words are shown in bigger and bolder font. Viewing the cloud, we can quickly spot the most used words in the log events to infer communicated topics. Since the execution events that are added in the latest release can only have one release of life-time, we remove them to avoid biasing the results.

Results

Figure 3 shows the tag clouds of long-lived and short-lived *CI* in *Hadoop*. We note the frequent occurrences of "mapred", which refers to the MapReduce library used by *Hadoop*.

Comparing both tag clouds, we find that the *long-lived CI* for *Hadoop* represents the "core" functionalities and high-level (domain-level) information about the software system. Such information exists throughout the history of *Hadoop*. For example, "tasktracker", "reduce" and "attempt" are the most important high-level concepts of the programming model and

TABLE VI
CI MODIFICATION PATTERNS AND EXAMPLES.

Pattern	**Definition**	**Example**	
		Before	**After**
Adding context	Additional context is added into 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 record 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 VII
DETAILED PERCENTAGES OF DIFFERENT PATTERNS OF CONTEXT MODIFICATIONS OF *Hadoop*.

Release	**Adding context**	**Deleting context**	**Redundant context**	**Rephrasing context**	**Merging context**	**Splitting context**
0.15.0	1.41	0	2.82	2.82	0	0
0.16.0	0	0	2.82	0	0	0
0.17.0	0	0	0	8.45	0	0
0.18.0	0	0	0	**28.17**	0	0
0.19.0	0	2.82	0	4.23	0	0
0.20.0	2.82	1.41	1.41	0	1.41	0
0.20.1	0	1.41	0	1.41	1.41	0
0.20.2	0	0	0	0	0	0
0.21.0	9.86	1.41	1.41	**16.9**	2.82	2.82

distributed computing platform of *Hadoop* [16]. An example of a *long-lived CI* is "TaskTracker Task task_id is done", which is recorded in *CI* whenever a small part of a distributed computing program is completed.

On the other hand, the *short-lived CI* typically corresponds to low-level (implementation-level) concepts. The most frequent word, "Java", is part of many error messages. For example, "ipc.RemoteException: java.io.IOException" is recorded in *CI* when *Hadoop* cannot perform inter-procedural communication between machines. Other low-level implementation details, such as "DELEGATINGMETHODACCESSORIMPL" and "NATIVEMETHODACCESSORIMPL", are also communicated for a short period of time.

Most *Log Processing Apps* are designed for recovering high-level information about the systems, e.g., system workload rather than implementation details. The *CI* of interest to such applications is relatively stable, leading to low maintenance cost for such applications. However, there are a few *Log Processing Apps* that are designed for debugging purposes. Such applications are much more fragile as their corresponding *CI* is continuously changing.

Long-lived CI *mainly corresponds to the domain-level information about the software systems, while short-lived* CI *tells more about low-level implementation details.*

V. 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., [17]–[19]), this paper studies the evolution of software systems from the perspective of non-code artifacts associated with these systems. For example, the evolution of the following non-code artifacts has been studied before:

TABLE VIII
DETAILED PERCENTAGES OF DIFFERENT PATTERNS OF CONTEXT MODIFICATIONS OF *EA*.

Release	Adding context	Deleting context	Redundant context	Rephrasing context	Merging context	Splitting context
0.1.1	0	0	0	0	0	0
0.1.2	6.78	1.69	0	**20.34**	0	0
0.1.3	**22.03**	0	0	10.17	0	1.69
0.1.4	6.78	0	0	1.69	0	0
0.1.5	8.47	1.69	0	0	0	0
0.1.6	1.69	0	0	1.69	0	0
0.2.0	6.78	0	0	3.39	1.69	0
0.2.1	0	0	0	3.39	0	0

- **Build Systems:** The build system decides which components should be built and incrementally builds the source code. Adams *et al.* [20], [21] and McIntosh *et al.* [22], [23] study different types of build systems, such as Makefile and Apache ANT. They find that build systems co-evolve with source code.
- **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.* [24] study the evolution of telephony software systems by studying the user documentation of telephony features in the phone books of Atlanta.
- **Menu Structure:** His *et al.* [25] study the evolution of Microsoft Word by looking at changes to its menu structure. Hou *et al.* [26] study the evolution of UI features in 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.* [27] propose a technique to evaluate the efficiency of software feature development by studying the evolution of call graphs generated during the execution of the same features. Greevy *et al.* [28] use program slicing to study the evolution of features.
- **Communicated Information:** We divide *CI* into *CI* about code and *CI* about the execution.
  - **Evolution of Code *CI*:** 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.* [29] 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.* [30], [31] study the evolution of code comments in 8 software projects.
  - **Evolution of Execution *CI*:** To the best of our knowledge, this paper is the first work that seeks to study the evolution of execution *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), but makes the applications, depending 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 [32] (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. 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.* [33], 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 implementation changes.

C. Traceability between Logs and Log Processing Apps

For most software developers, logs are considered 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 one of the first studies to explore how changes in parts of an ecosystem (communicated information, i.e., logs) might impact other parts of the system (*Log Processing Apps*). The need for such types of studies was noted by Godfrey and German [34], as they recognized that most software systems today are linked in a formal or informal manner with other systems within their ecosystems.

Our study provides evidence of Lehman's earlier work [19], which notes 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 ecosystem, namely the system, the *CI* by the systems, 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.

VI. Threats to Validity

This section presents the threats to validity of our study.

Internal 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 are needed to see whether our findings can generalize. The studied logs are collected from specific workloads, which may not generalize. We plan to study in-field execution logs in our future work.

External validity

Our study has several manual steps, including the checking of log modifications, the classification of log reformatting and study of the characteristic of *long-lived CI*. Our findings may contain subjectivity bias.

Construct validity

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

Our study is mainly based on the abstraction of execution events proposed by Jiang *et al.* [12]. This approach is shown to have high precision and we customize the approach to better fit the two subject systems, but fault 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 false abstracted execution events in our study.

VII. Conclusion

Communicated information, such as execution logs, error messages and comments, is generated by snippets of code inserted explicitly by domain experts to record valuable execution information. *Log Processing Apps* are developed to analyze such valuable information to assist in software testing, system monitoring and program comprehension. *Log Processing Apps* highly depend on *CI* and are hence impacted by changes to the *CI*. In this paper, we performed an exploratory study on the *CI* from 10 releases of an open source software named *Hadoop* and 9 releases of a legacy enterprise application. The goal of this paper is to understand the evolution of *CI* as well as the impact of such evolution on the maintenance of the corresponding *Log Processing Apps*.

Our study shows that systems communicate more about their execution as they evolve. We find that 20% to 30% of *CI* changes leads to negative and unnecessary changes to *Log Processing Apps*. Our results also show that the long-lived *CI* mainly corresponds to high-level domain concepts, while short-lived *CI* mainly corresponds to the system implementation-level concepts.

Our results indicate that additional maintenance resources should be allocated to maintain *Log Processing Apps*, when major changes are introduced into the software systems. In particular, modifications to communicated context are likely to impact the *Log Processing Apps*, and cause them to fail. In addition, more resources should be allocated to maintain *Log Processing Apps* designed for debugging problems (*from short-lived CI*) than *Log Processing Apps* designed for recovering high-level system information (from *long-lived CI*).

Our immediate future work concentrates on studying how the *Log Processing Apps* react to the identified evolution characteristics of *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*. New York, NY, USA: 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*. Piscataway, NJ, USA: 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*. Berkeley, CA, USA: 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*, 2009, pp. 125–134.

[7] "Infosphere streams," http://www-01.ibm.com/software/data/infosphere/streams/.

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

[11] 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*. New York, NY, USA: ACM, 2007, pp. 199–211.

[12] 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.

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

[14] 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.

[15] "Tag cloud," http://www.wordle.net.

[16] T. White, *Hadoop: The Definitive Guide*. Oreilly & Associates Inc, 2009.

[17] 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*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 160–166.

[18] 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*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 131–.

[19] 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*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 20–32.

[20] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the linux build system," *Electronic Communications of the ECEASST*, vol. 8, February 2008.

[21] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter, "Design recovery and maintenance of build systems," in *ICSM '07: Proceedings of the 23rd International Conference on Software Maintenance*, L. Tahvildari and G. Canfora, Eds. Paris, France: IEEE Computer Society, October 2007, pp. 114–123.

[22] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ant build systems," in *MSR '10: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, Cape Town, South Africa, May 2010, pp. 42–51.

[23] S. McIntosh, B. Adams, Y. Kamei, T. Nguyen, and A. E. Hassan, "An Empirical Study of Build Maintenance Effort," in *ICSE '10: Proceedings of the 33rd International Conference on Software Engineering*, Waikiki, Honolulu, Hawaii, May 2011, to appear.

[24] 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*, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 421–430. [Online]. Available: http://portal.acm.org.proxy.queensu.ca/citation.cfm?id=381473.381517

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

[26] 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*. New York, NY, USA: ACM, 2009, pp. 122–135.

[27] 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*, 2008, pp. 7–16.

[28] 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.

[29] 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*. New York, NY, USA: ACM, 2006, pp. 179–180.

[30] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 70–79.

[31] 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.

[32] M. W. Johnson, "Monitoring and diagnosing application response time with arm," in *Proceedings of the IEEE Third International Workshop on Systems Management*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 4–.

[33] 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*. New York, NY, USA: ACM, 2011, pp. 3–14.

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