Assisting Developers of Big Data Analytics Applications When Deploying on Hadoop Clouds

Weiyi Shang[†], Zhen Ming Jiang[†], Hadi Hemmati[†], Bram Adams[‡], Ahmed E. Hassan[†], Patrick Martin[§]
[†]Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, Canada
[‡]Département de Génie Informatique et Génie Logiciel, Polytechnique Montréal, Montréal, Québec, Canada
[§]Database Systems Laboratory, School of Computing, Queen's University, Kingston, Canada
{swy, zmjiang,hhemmati, martin, ahmed}@cs.queensu.ca, bram.adams@polymtl.ca

***Abstract*—Big data analytics is the process of examining large amounts of data (big data) in an effort to uncover hidden patterns or unknown correlations. Big Data Analytics Applications (BDA Apps) are a new type of software applications, which analyze big data using massive parallel processing frameworks (e.g., Hadoop). Developers of such applications typically develop them using a small sample of data in a pseudo-cloud environment. Afterwards, they deploy the applications in a large-scale cloud environment with considerably more processing power and larger input data (reminiscent of the mainframe days). Working with BDA App developers in industry over the past three years, we noticed that the runtime analysis and debugging of such applications in the deployment phase cannot be easily addressed by traditional monitoring and debugging approaches.**

In this paper, as a first step in assisting developers of BDA Apps for cloud deployments, we propose a lightweight approach for uncovering differences between pseudo and large-scale cloud deployments. Our approach makes use of the readily-available yet rarely used execution logs from these platforms. Our approach abstracts the execution logs, recovers the execution sequences, and compares the sequences between the pseudo and cloud deployments. Through a case study on three representative Hadoop-based BDA Apps, we show that our approach can rapidly direct the attention of BDA App developers to the major differences between the two deployments. Knowledge of such differences is essential in verifying BDA Apps when analyzing big data in the cloud. Using injected deployment faults, we show that our approach not only significantly reduces the deployment verification effort, but also provides very few false positives when identifying deployment failures.

***Index Terms*—Big-Data Analytics Application, Cloud Computing, Monitoring and Debugging, Log Analysis, Hadoop**

I. Introduction

Big Data Analytics Applications (BDA Apps) are a new category of software applications that leverage large-scale data, which is typically too large to fit in memory or even on one hard drive, to uncover actionable knowledge using large-scale parallel-processing infrastructures [1]. The big data can come from sources such as runtime information about traffic, tweets during the Olympic games, stock market updates, usage information of an online game [2], or the data from any other rapidly growing data-intensive software system. For instance, EBAY [1] has deployed BDA Apps to optimize the search of products by analyzing over 5 PBs data using more than 4,000 CPU cores [3].

[1]www.ebay.com last checked Feburary 2013.

Over the past three years we have been working closely with BDA App developers in industry. We noted and found that developing BDA Apps brings many new challenges compared to traditional programming and testing practices. Among all challenges in different phases of BDA App development, the deployment phase introduces unique challenges related to verifying and debugging the BDA executions, as BDA App developers want to know if their BDA App will function correctly once deployed. Similar observations were recently noted in an interview of 16 professional BDA App developers at Microsoft [1].

In practice, the deployment of BDA Apps in the cloud follows these three steps: 1) developers implement and test the BDA App in a small or pseudo cloud (using virtual or physical machines) environment using a small data sample, 2) developers deploy the application on a larger cloud with a considerably larger data set and processing power to test the application in a real-life setting, and 3) developers verify the execution of the application to make sure all data are processed and all jobs are successful. The traditional approach for deployment verification is to simply search for known error keywords related to unusual executions. However, such verification approaches are very ineffective in large cloud deployments. For instance, a common basic approach for identifying deployment problems is searching for "killed" jobs in the generated execution logs (the output of the internal instrumentation) of the underlying platform hosting the deployed application [4]. However, a simple keyword search would lead to false positive results since a platform such as Hadoop may intervene in the execution of a job, kill it and restart it elsewhere to achieve better performance, or it might start and kill speculative jobs [4]. Considering the large amount of data and logs, such false positives rapidly overwhelm the developer of BDA Apps.

In this paper, we propose an approach for verifying the runtime execution of BDA Apps after deployment. The approach abstracts the platform's execution logs from both the small (pseudo) and large scale cloud deployments, groups the related abstracted log lines into execution sequences for both deployments, then examines and reports the differences between the two sets of execution sequences. Ideally, these two sets should be identical for a successful deployment. However, due to framework configurations and data size differences, the

underlying platform may execute the applications differently. Among the delta sets of execution sequences between these two sets, we filter out sequences that are due to well-known platform-related (in our case study Hadoop) differences. The remaining sets of sequences are potential deployment failures/anomalies that should be reported and carefully examined.

We have implemented our approach as a prototype tool and performed a case study on three representative Hadoop [4] BDA Apps. The choice of Hadoop is due to it being one of the most used platforms for Big Data Analytics in industry today. However, our general idea of using the underlying platform's logs as a means for BDA App monitoring in the cloud, is easily extensible to other platforms, such as Microsoft Dryad [5]. The case study results show that our log abstraction and clustering into execution sequences not only significantly reduces the amount of logs (by between 86% to 97%) that should be verified, but it also provides much higher precision for identifying deployment failures/anomalies compared to a traditional keyword search approach (commonly used in practice today). In addition, practitioners who have used our approach in practice have noted that the reporting of the abstracted execution sequences, rather than raw log lines, provides a summarized context that dramatically improves their efficiency in identifying and investigating failure/anomaly.

The rest of this paper is organized as follows. We present a motivating example in Section II. We present *Hadoop*, the platform that we studied in Section III. We present our approach to summarize logs into execution log sequences in Section IV. We present the setup for our case studies in Sections V. We present the results of our case study in Section VI. We discuss other features of our approach in Section VII and discuss the limitations of our approach in Section VIII. We present prior work related to our approach in Section IX. Finally, we conclude the paper in Section X.

II. A Motivating Example

We now present a hypothetical but realistic motivating example to better explain the challenges of deploying BDA Apps in a cloud environment.

Assume developer Ian developed a BDA App that analyzes the user information from a large-scale social network. Ian has thoroughly tested the App on an in-house small-scale cloud environment with a small sample of testing data. Before officially releasing the App, Ian needs to deploy the App in a large-scale cloud environment and run the App with real-life large-scale data. After the test run of the App in the real cloud setup, Ian needs to verify whether the App behaves as expected or not, in the testing environment.

Ian followed a traditional approach to examine the behaviour of the App in the cloud environment. He leveraged the logs from the underlying platform (e.g., Hadoop) to find whether there are any problematic log lines. After downloading all the logs from the cloud environment, Ian found that the logs are of enormous size because the cloud environment contains thousands of nodes and the processed real-life data is in PB scale, which makes the manual inspection of the logs impossible. Therefore, Ian performed a simple keyword search on the logs. The keywords are based on his own experience of developing BDA Apps. However, the keyword search still returns thousands of problematic log lines. By manually exploring the problematic log lines, Ian found that a large portion of the log lines do not indicate any problematic executions (i.e., false positives). For example, the run-time scheduler of the underlying platform often kills remote processes and restarts them locally to achieve better performance. However, such kill operations lead to seemingly problematic logs that are retrieved by his keyword search. Moreover, for each log line, Ian must trace through the log files across multiple nodes to gain some context about the generated log files (and in many instances he discovers that such log lines are expected and are not problematic ones). In short, identifying deployment problems of the BDA App is excessively difficult and time consuming. Moreover, this difficulty increases considerably as the size of the analyzed data grows and the size of the cloud increases.

From the above example, we observe that verifying the deployment of BDA Apps in a cloud environment with large-scale data is challenging. Although today, developers primarily use grep [6] to locate possible troublesome instrumentation logs, uncovering the related context of the troublesome logs is still challenging with enormously large data (as noted in recent interviews of BDA App developers [1]).

In the following sections, we present our approach, which summarizes the large amount of platform logs and presents them in tables where developers can easily note troublesome events and where they are able to easily view such events in the context of their execution (since the table shows summarized execution sequences).

III. Large-scale Data Analysis Platforms: Hadoop

Hadoop is one of the most widely used platforms for the development of BDA Apps in practice today. We briefly present the programming model of Hadoop, then present the Hadoop logs that we use in our case studies.

A. The MapReduce Programming Model

Hadoop is an open-source distributed platform [4] that is supported by Yahoo! and is used by Amazon, AOL and a number of other companies. To achieve parallel execution, Hadoop implements a programming model named MapReduce. This programming model is implemented by many other cloud platforms as well [5], [7].

MapReduce [8] is a distributed divide-and-conquer programming model that consists of two phases: a massively parallel "Map" phase, followed by an aggregating "Reduce" phase. The input data of MapReduce is broken down into a list of key/value pairs. Mappers (processes assigned to the "Map" phase) accept the incoming pairs, process them in parallel and generate intermediate key/value pairs. All intermediate pairs having the same key are then passed to a specific Reducer (process assigned to the "Reduce" phase). Each Reducer performs computations to reduce the data to one

single key/value pair. The output of all Reducers is the final result of a MapReduce run.

To illustrate MapReduce, we consider an example MapReduce process that counts the frequency of word lengths in a book. Mappers take each single word from the book and generate a key/value pair of the form "word length/dummy value". For example, a Mapper generates a key/value pair of "5/hello" from the input word "hello". Afterwards, the key/value pairs with the same key are grouped and sent to Reducers. Each Reducer receives the list of all key/values pairs for a particular word length and hence can simply output the size of this list. If a reducer receives a list with key "5", for example, it will count the number of all the words with length "5". If the size is *n*, it generates an output pair "5/n" which means there are *n* words with length "5" in the book.

B. Components of Hadoop

Hadoop has three types of execution components. Each component has logging enabled in it. Such platform logging tracks the operation of the platform itself (i.e., how the platform is orchestrating the MapReduce processing). Today, such logging is enabled in all deployed Hadoop clusters and it provides a glimpse into the inner working mechanism of the platform itself. Such inner working mechanism is impacted by any problems in the cloud on which the platform is executing. The three execution components and a brief example of the logs generated by them are as follows:

- **Job.** A Hadoop program consists of one or multiple MapReduce steps running as a pipeline. Each MapReduce step is a Job in Hadoop. A JobTracker is a process initialized by the Hadoop platform to track the status of the Jobs. The information tracked by the JobTracker includes the overall information of the Job (e.g., input data size) and the high-level information of the execution of the Job. The high-level information of the Job's execution corresponds to the executions of Map and Reduce. For example, a Job log may say that "the Job is split into 100 Map Tasks" and "Map TaskId=id is finished at time t1".
- **Task.** The execution of a Job is divided into multiple Tasks based on the MapReduce programming model. Therefore, a Task can be either a Map Task that corresponds to the Map in the *MapReduce* programming model, or a Reduce Task. The Hadoop platform groups a set of Map or Reduce executions together to create a Task. Therefore, each Task contains more than one execution of Map or Reduce. Similar to the JobTracker, the TaskTracker monitors the execution of a Task. For example, a Task log may say "received commit of Task Id=id".
- **Attempt.** To support fault tolerance, the Hadoop platform allows each Task to have multiple trials of execution. Each execution is an Attempt. Typically, only when an Attempt of a Task has failed, another Attempt of the same Task will start. This restart process continues until the Task is successfully completed or the number of failed Attempts is larger than a threshold. However, there are exceptions, such as "speculative execution", which we discuss later in this paper. The attempt is also monitored by the TaskTracker and the detailed execution information of the Attempt, such as "Reading data for Map task with TaskID=id", is recorded in the Attempt logs.

The Job, Task and Attempt logs form the source of information used by our approach. We use the former kinds of logs instead of application-level logs since such logs provide information about the inner working of the platform itself, and not the application, which is assumed to be correctly implemented for our purposes. In particular, the platform logs provide us with information about any deployment problems.

IV. Approach

The basic idea behind our approach is to cluster the platform logs to improve their comprehensibility, and to help understand and flag differences in the run-time behaviour.

As mentioned before, our approach is based on the analysis of platform logs of BDA Apps. These logs are generated by the statements embedded by the platform developers because they consider the information to be particularly important. Containing rich knowledge, but not fully explored, platform logs typically consist of the major system activities and their associated contexts (e.g., operation ids). The log is a valuable resource for studying the run-time behaviour of a software system, since they are generated by the internal instrumentations and are readily available. However, previous research shows that logs are continuously changing and evolving [9]. Therefore, ad hoc approaches based on keyword search may not always work. Thus we propose an approach that does not rely on particular phrases or format of logs. Figure 1 shows an overview of our approach.

Our approach compares the run-time behaviour of the underlying platform of BDA Apps in testing environment with a small testing data sample to the cloud environment with large-scale data. To overcome the enormous amount of logs generated by a BDA platform and to provide useful context for the developers looking at our results, we recover the execution sequences of the logs.

A. Execution Sequence Recovery

In this step, we recover sequences of the execution logs. The log sequence clustering includes three phases.

1) Log Abstraction: Log files typically do not follow strict formats, but instead contain significant unstructured data. For example, log lines may contain the task type, the execution time stamp and a free form – making it hard to extract any structured information from them. In addition to being in free form, log lines contain static and dynamic information. The static information is specific to each particular event while the dynamic values of the logs describe the event context. We use a technique proposed by Jiang *et al*. [10] to abstract logs. This technique is designed to be generalizable as it does not rely on any log formats. Using the technique, we first identify the static and dynamic values of the logs based on a small sample of logs. Then we apply the identified static and dynamic parts

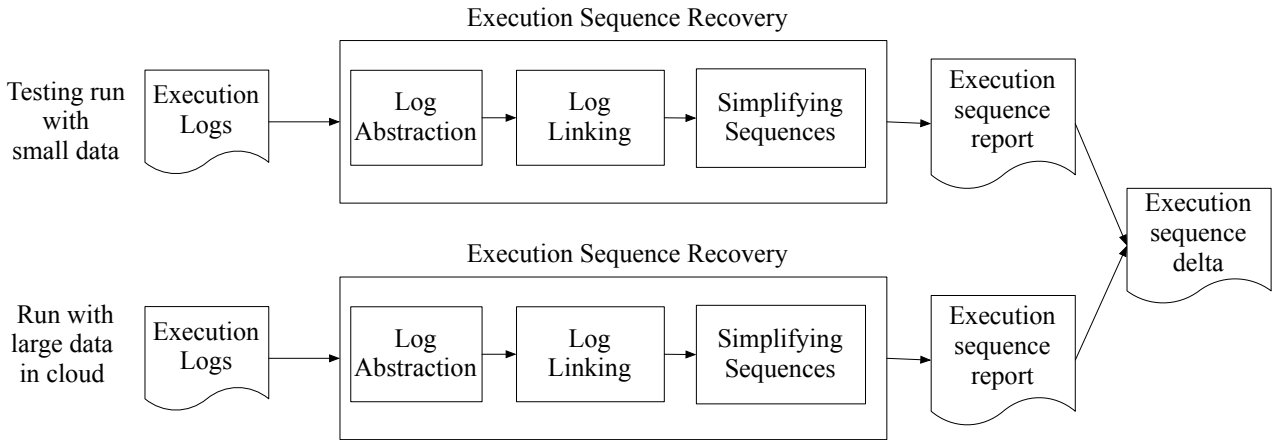
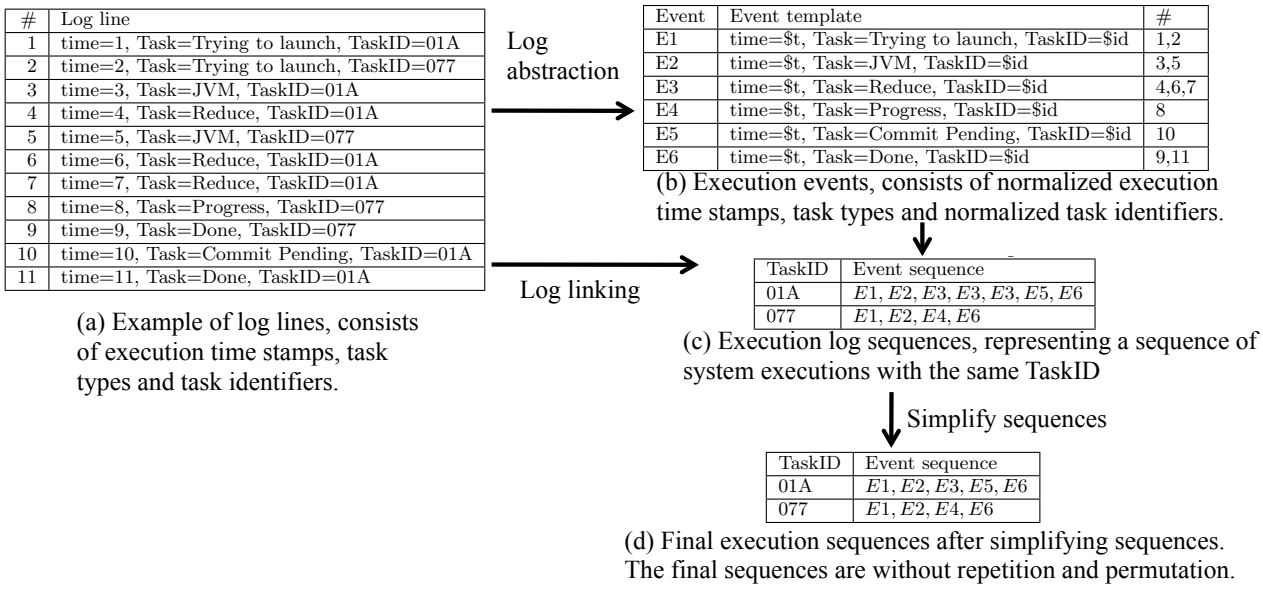


Fig. 1. Overview of our approach.



#	Log line
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

(a) Example of log lines, consists of execution time stamps, task types and task identifiers.

Log abstraction

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

(b) Execution events, consists of normalized execution time stamps, task types and normalized task identifiers.

Log linking

TaskID	Event sequence
01A	$E1, E2, E3, E3, E3, E5, E6$
077	$E1, E2, E4, E6$

(c) Execution log sequences, representing a sequence of system executions with the same TaskID

Simplify sequences

TaskID	Event sequence
01A	$E1, E2, E3, E5, E6$
077	$E1, E2, E4, E6$

(d) Final execution sequences after simplifying sequences. The final sequences are without repetition and permutation.

Fig. 2. An example of our approach for summarizing the run-time behaviour of BDA Apps.

on full logs to abstract the logs. Figure 2 shows an example of a log file with 11 log lines and how we process it. Each log line contains the execution time stamp, the task type, and the task ID. The log lines are abstracted into six different system events, as shown in Figure 2-b. The "$id" and "$t" identifiers indicate two dynamic values.

2) Log Linking: This phase uses dynamic values, such as "$id", to link log lines into a sequence. The linking heuristic is based on the dynamic values. In our example, TaskID is used for log linking since TaskID represents some kind of session "ID". Therefore, line 1 and line 3 in the input data in Figure 2-a can be linked together since they contain the same TaskID. Similar to log abstraction, we also identify the linkage among a few IDs based on a small sample of data, then apply the linking on full data.

Figure 2-c shows the resulting sequences after abstracting the logs and linking them into sequences using the TaskID values. Events E1, E2, E3, E5 and E6 are linked together (note that event E3 has been executed three times) and events E1, E2, E4, E6 are linked together since the same TaskID values are shared, among them.

3) Simplifying Sequences: An example of repetition is sequence caused by loops. For example, for sequences about reading data from a remote node, there would be repeated events about fetching the data. Without this step, similar log sequences that include different occurrences of the same event are considered different sequences, although they indicate the same system behaviour in essence. These repeated events need to be suppressed to improve the readability of the generated summaries. Therefore, we use regular expression techniques to detect and suppress the repetitions. For the example shown in Figure 2, our technique detects the repetition of E3 in the sequence "E1, E2, E3, E3, E3, E5, E6", and reduces this sequence to "E1, E2, E3, E5, E6".

The second step of simplifying sequences is dealing with permutations of sequences. The reason why permutations oc-

cur is that sometimes the events execute asynchronously on the distributed computing platforms, although the corresponding sequences result in the same system behaviour. We group the permutations of a sequence together to simplify the sequences. For example, if we recovered two sequences "E1, E2, E3, E4" and "E1, E3, E2, E4", we would group these two sequences together in this step.

After simplifying sequences, we obtain the final log sequences in Figure 2-d.

B. Generating Reports

We generate a log sequence report in HTML format. Figure 3 shows an example report. The report consists of two parts: an overview of the number of execution log sequences and a list of sample log lines. To ease the comparison of different reports, each event is represented by the same unique number across the reports. An exemplar sequence from the analyzed logs is shown in each row of the report to provide developers an actual example with a realistic context.

V. Case Study

In this section, we present the design of the case study that we performed to evaluate our approach.

A. Subject Applications

We use three BDA Apps as subjects for our study. Two out of the three applications are chosen to be representative of industrial BDA Apps. In addition, to avoid potential bias from the development of the applications, we chose one application that is developed from scratch and another application that is re-engineered by migrating a set of Perl scripts to the Hadoop platform. In addition, to ease the replication of our approach by others, we chose a third application from *Hadoop*'s official example package. The overview of the three BDA Apps is shown in Table I.

- **WordCount.** *WordCount* is an application that is released with *Hadoop* as one of the examples of MapReduce programming. The *WordCount* application analyzes the input files and counts the number of occurrences of each word in the input files.
- **PageRank.** *PageRank* [11] is a program used by the Google Internet search engine for rating Web pages. We implemented the *PageRank* algorithm on *Hadoop*.
- **JACK.** *JACK* is an industrial application that uses data mining techniques to identify problems in load tests [12]. This tool is used in practice on a daily basis. We migrated *JACK* to the *Hadoop* platform.

Note that none of the three above BDA Apps have their own logs and the logs that our approach uses are the platform (Hadoop) logs generated during the execution of these applications.

B. The Experiment's Environment Setting

As input data for *WordCount* and *JACK*, we use two groups of execution log files of a large enterprise application. The input data for the *PageRank* application, however, comes from two social-network datasets from the *Stanford Large Network Dataset Collection*[2]. Table II summarizes the overall size of the input data for the studied applications.

TABLE I
Overview of the three subject BDA Apps.

	WordCount	PageRank	JACK
Source	Hadoop: Official Example	Google: Developed from scratch	RIM: Migrated from Perl
Domain	File processing	Social network	Log analysis
Injected problem	Machine failure	Missing supporting library	Lack of disk space

TABLE II
Overview of the BDA App's input data size.

	WordCount & JACK	PageRank
Large Data	$3.69GB$	$1.08GB$
Small Data	$597MB$	$10.7MB$

To have a proof of concept experiment setting, we performed our experiments on an in-house small cloud (a cluster with 40 cores across five machines). Each machine has Intel Xeon E5540 (2.53GHz) with 8 cores, 12 GB memory, a Gigabit network adaptor and SATA hard drives. The operating system of the machines is Ubuntu 9.10.

VI. Case Study Results

In this section, we present our research questions, and the results of our case study. For each research question, we present the motivation of the question, our approach to answer the question, and the results.

A. RQ1: How Much Effort Reduction Does Our Approach Provide, When Verifying the Deployment of BDA APPs in the Cloud?

Motivation

Developers often use simple text search techniques to identify troublesome events when deploying BDA Apps. For example, keywords such as "kill" and "fail" are often used to find problematic tasks in Hadoop. Due to the decision by underlying platform (e.g. algorithms that Hadoop uses for assigning tasks to machines), a problematic event might be caused by some other reasons than an actual deployment failure. Two commonly seen examples of such reasons on Hadoop platform are "Task exceptions" and "Speculative execution":

- **Task exceptions.** When there is an exception during the execution of a Hadoop task, the task will be killed and restarted on another cloud node. Therefore, a keyword search for "kill" on the log lines would flag such Hadoop decisions as a sign of failure, even though this is supposed to be transparent from the developer.
- **Speculative execution.** The overall performance of a Hadoop Job may slow down because of some slow-running tasks. To optimize the performance, Hadoop

[2]http://snap.stanford.edu/data/ last check Aug, 2012.

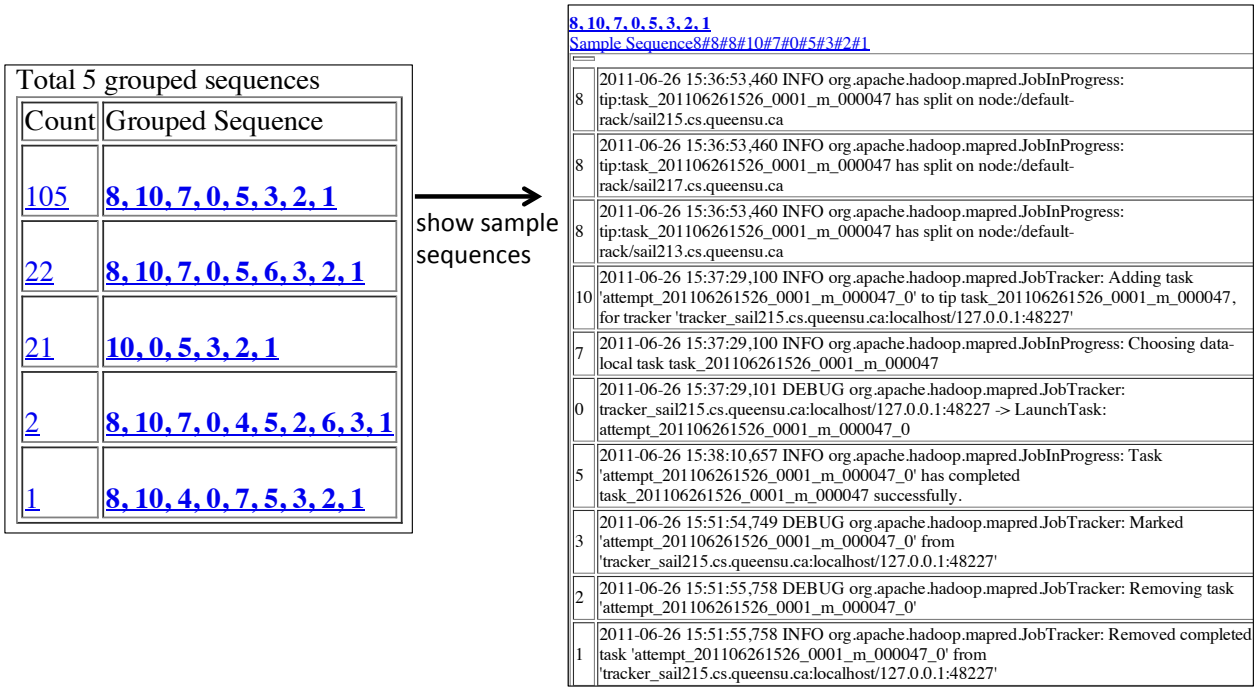


Fig. 3. An example of our log sequences report.

replicates the unfinished tasks on idle machines. When one of the replicated tasks, or the original task, is finished, Hadoop commits the results from the task and kills other replicas. This mechanism is similar to the *Backup Task* in Google's MapReduce platform [8]. The replication and killing are decided at run-time and are not signs of deployment failures. However, again, a keyword search would flag them as a problem to be verified.

Therefore, a simple text search for such keywords may result in a very large set of irrelevant log lines for manual verification. In this research question, we investigate whether our approach saves any effort in the cloud deployment verification process.

Approach

To evaluate our approach in terms of effort reduction, we use the amount of log lines that must be examined as a basic approximation of the amount of effort. We first use the traditional (most-often-used in practice today (e.g., [1])) approach of searching for keywords in the raw log lines as a baseline of comparison. The keywords that we use in this experiment are common basic keywords ("kill", "error", "fail", "exception" and "died") that are usually a sign of failure in a log line. We applied this search on all three BDA Apps. We measure the number of log lines with these keywords as the baseline effort.

To apply our approach for deployment verification, we first recover execution sequences of the three BDA Apps, when deployed on a cloud environment. We then compare the two sets of log sequences (small-scale environment and large cloud) and identify the *delta set* (the execution sequences without exact match). The last step involves searching for the same keywords as the traditional approach to measure the number of execution sequences and log events that are required to examine.

TABLE III
EFFORT REQUIRED TO VERIFY THE CLOUD DEPLOYMENT USING OUR APPROACH VERSUS THE TRADITIONAL KEYWORD SEARCH.

	Using our approach		**Using keyword search**
	# execution sequences	# unique log events	#log line with keyword
WordCount	19	64	467
PageRank	55	83	1,739
JACK	20	67	726

TABLE IV
REPEATED EXECUTION SEQUENCES BETWEEN RUNNING THE BDA APPS ONCE, TWICE AND THREE TIMES.

	once and twice (%)	twice and three times (%)
WordCount	98.4	99.1
PageRank	95.0	95.1
JACK	99.5	99.8

Results

The results from Table III show that with our approach the number of execution log sequences (and their corresponding number of log events) to verify is 19(64), 55(83), and 20(67) for WordCount, PageRank, and JACK respectively. However, the number of raw log lines to verify after the keyword search, i.e., the traditional approach, is 467, 1739, and 726 for WordCount, PageRank, and JACK respectively. Therefore, our approach provides 86%, 95%, and 97% effort reduction over

TABLE V
NUMBER OF LOG LINES GENERATED BY RUNNING BDA APPS ONCE, TWICE AND THREE TIMES.

	once	twice	three times
WordCount	78 K	309 K	393 K
PageRank	109 K	217 K	474 K
JACK	237 K	419 K	666 K

the traditional approach, ignoring the fact that verifying a log line may require more effort than verifying a log event. Indeed, verifying a log line requires checking the other log lines to get a context of the failure, whereas the log events are already shown in the context (i.e., the execution sequences). Also, notice that our approach does not incur any instrumentation overhead since the platform logs are already available.

Another interesting point is that when the input data grows, several new execution sequences and log lines appear. That is due to the fact that the behaviour is not present with the smaller runs. However, when moving to bigger runs, the execution sequences will not increase dramatically. The reason is that most of the runs, in the abstract level, are identical. Therefore, the size of the final sequences to verify will show very minor increases. However, the log lines to be verified using traditional approach always increase proportional to the data. Table IV shows the number of repeated execution sequences when running the same BDA App once, twice, and three times. In Table V, we report the number of log lines to be verified using the traditional approach for the same BDA App executions. The large portion of repeated execution sequences in the number of execution sequences vs. the rapid growth in the number of log lines emphasizes the effectiveness of our approach in terms of effort reduction during verification of the deployment of a BDA App in the cloud.

Our approach reduces the verification effort by between 86% to 97% when verifying the cloud deployment of BDA Apps.

B. RQ2: How Precise and Informative is Our Approach When Verifying Cloud Deployments?

Motivation

As discussed in RQ1, a flagged sequence (using our approach) or a flagged log line (using the traditional approach), might be caused by some other reasons than an actual deployment failure. We consider such flagged sequences or log events, e.g., those that are related to " Task exception" and "Speculative execution", as false positive results that affect the precision of the approaches. Therefore, in this question, we compare the two approaches in terms of precision. We also discuss how our approach facilitates the verification of the flagged execution sequences.

Approach

In this research question, we categorize the flagged logs/execution sequences by the traditional/our approach into two classes: actual failures (true positives) and platform-related events (false positives). To get the instances of the actual failure, we intentionally injected three different failures, which are commonly observed by BDA App developers [13], into our experimental environment, during the execution of the three subject programs. These three failures are all encountered often in our real-life experience of using Hadoop in the large industrial clouds. The three injected failures are as follows:

1) **Machine failure.** Machine failure is one of the common system errors in distributed computing. To inject this failure, we manually turn off one machine in the cluster.
2) **Missing supporting library.** A cluster administrator may decide to expand the size of the cluster. However, the new machines in the cluster may miss supporting libraries or the versions of the supporting libraries may be outdated. We inject this failure by removing one required library of the analysis.
3) **Lack of disk space.** Disks often run out of space while a BDA App is running on the platform due to the large amount of generated intermediate data. We manually fill up one of the machine's disks to inject this failure.

Next, we manually analyzed the log sequences in the HTML reports and identified any false positive instances.

Results

Table VI summarizes the number of false positives, total number of flagged sequences/log lines, and the precision of both approaches. The precision of our approach is 21, 38, and 10% for WordCount, PageRank, and JACK respectively. The range of the number of false positive sequences to verify is 15 to 34 sequences (16-49 log events). However, the precision of the traditional approach is 7, 84, and 10% for WordCount, PageRank, and JACK respectively, while the range of the number of false positive log lines to verify is 432 to 650 log lines. A more detailed analysis of the results shows that the only case where the precision of our approach is outperformed by traditional approach is with JACK BDA App where one single exception is appearing in almost every log line. Though the traditional approach is of higher precision, however, the same exception produces 1,467 log lines that must be examined each by hand to determine their context and decided whether they are problematic or not. Unfortunately, since a keyword approach does not provide any abstraction all log lines would need to be examined carefully, even though they all are instances of the same abstract problem.

Note that the recall for both approaches is 100%, since all instances of log lines and execution sequences related to the failures are identified by the keyword search. However, there are cases that deployment failure might not be possible to catch by a keyword search. For example, a temporary network congestion may cause the pending queue to be very long, but logs may record that the pending queue is too long without making use of an "error" like string in the log line. In some cases, a node in the cloud may even fail without recording any error message [14]. In such situations, our approach is even superior, since the traditional approach simply would not work (as no "error" log lines are produced) and the developer would miss such problems all together unless he or she examines

TABLE VI
NUMBER OF FALSE POSITIVES, TRUE POSITIVES, AND THE PRECISION OF BOTH OUR APPROACH AND THE TRADITIONAL KEYWORD SEARCH.

	Using our approach			**Using keyword search**		
	false positive	true positive	precision	false positive	true positive	precision
WordCount	15	4	21%	432	35	7%
PageRank	34	21	38%	272	1467	84%
JACK	18	2	10%	650	76	10%

each log line. However, our recovered execution sequences still would work, since it only depends on finding the delta set of sequences when switching from the small to large cloud.

Another interesting aspect of our approach, which was the initial motivation of this work, is the extra information (context of a log line) that our approach provides for deployment verification. Even after all the reduction that is performed by the keyword search approach, 467 to 1,739 log lines should be verified manually. As discussed earlier, there are false positives in the flagged log lines. Distinguishing them from true positives requires knowledge about the context of each line (otherwise both categories contain the failure-related keyword). Our approach provides such context by grouping the log events in one execution sequence, which speeds up the understanding and verification of the event.

The precision of our approach for assisting deployment verification of BDA Apps in the cloud is comparable with the precision of the traditional approach. However, our approach provides additional context information (execution sequences) that is essential in speeding up the manual investigation of flagged problems.

VII. DISCUSSION

In this section, we discuss other possible features of our approach. In particular, one feature is to support developers to understand the runtime differences when migrating BDA Apps from one platform to another.

To find the most optimal and economical platform for BDA Apps, a BDA App may need to be migrated from one Big Data Analytics platform to another [1]. This type of redeployments requires similar verifications as discussed in the research questions. Therefore, developers need an approach to help them in identifying any run-time behaviour change, caused by the migration. Identifying the differences between the execution of the BDA App in the two environments help verifying the new deployment and flag any potential failure or anomalies.

To assess the ability of our approach for identifying the potential redeployment problems in the cloud, we migrated the *PageRank* program from Hadoop to Pig platform. Pig [15] is a Hadoop-based [4] platform designed for analysis of massive amounts of data. To reduce the effort of coding in the MapReduce paradigm, Pig provides a high-level data processing language called Pig Latin [15]. Using Pig Latin, developers can improve their productivity by focusing on the process of data analysis instead of writing the boiler-plating MapReduce coding [16].

We ran PageRank three times on Hadoop and three times on Pig. After examining the sequence reports we could note the following differences between both platforms:

1. Hadoop-based PageRank Has More MapReduce Steps than Pig-based PageRank

In our implementation, the Hadoop-based PageRank consists of four MapReduce steps, while the Pig-based PageRank has eight lines of Pig scripts (each line is one step in the pipeline of the data analysis). However, in the real-life execution, the Pig platform groups the eight steps of data process into three MapReduce steps. Since the Hadoop platform does not have such a feature of grouping MapReduce steps, the execution of Hadoop-based PageRank has four MapReduce steps, as it is written.

2. Hadoop-based PageRank Has More Tasks than Pig-based PageRank

We examine the distribution of sequences in both implementations of PageRank. The results show that the total number of log sequences from the Hadoop-based PageRank is much larger than the Pig-based PageRank. For example, one run of the Hadoop-based PageRank generates, in total, over 700 execution log sequences in the Task log, while this number by the Pig-based PageRank is less than 30. This result indicates that the Pig-based PageRank splits the execution into a significantly smaller number of tasks than the Hadoop-based one. The reason is that based on the Hadoop instructions [4], the number of Map Tasks should be set to a relatively large number. Therefore, in our case study, the number of Map Tasks is configured to 200. However, Pig optimizes the platform configurations at run-time and reduces the number of Map Tasks to a smaller number to get better performance.

Identifying such differences would be extremely difficult by only looking at the raw log lines. Thus our approach not only assists developers of BDA Apps with the first deployment in the cloud but also helps them with any redeployment. We do note that our approach only works when the new platform is a derivative of the older platform (e.g., in the case of Pig, it provides a high-level abstraction to create programs that eventually still run on MapReduce. Hence we can compare the eventual MapReduce execution for the Pig program against the old MapReduce execution).

VIII. LIMITATIONS AND THREATS TO VALIDITY

We present the limitations and threats to validity for our approach in this section.

A. External Validity

As a proof of concept, we illustrate the use of our approach to address the challenges encountered in our experience. However, there are still other challenges of developing and testing BDA Apps, such as choosing an architecture that optimizes for cost and performance [1]. Our approach may not be able to address other challenges. Additional case studies are needed to better understand the strengths and limitations of our approach.

We only demonstrate the use of our approach on Hadoop, one of the most widely adopted underlying frameworks for BDA Apps, with three injected failures. In practice, we have tried our approach on several commercial BDA Apps on other underlying platforms. The only effort for adapting our approach to other platforms of BDA Apps is to determine the parameters for abstracting and linking the platform logs. Additional studies on other open source and commercial platforms with other types of failures are needed to study the generalizability of our approach.

All our experiments are carried out on a small-scale private experimental cluster, which mimics the large cloud with 40 cores. However, a typical environment for BDA Apps has more than 1,000 cores, such as Amazon EC2 [17]. The logs of such large-scale clouds do lead to considerably more logs and more sequences. From our experiences using our approach in practice on such large clouds, we have found that our approach performs even better than grep since the abstraction and sequencing leads to a drastic reduction in the amount of data. Such observation was noted in our case study as well. One interesting note is to support such large clouds we needed to re-implement our own approach to run in a cloud setting using the Hadoop platform (since we needed to process a very large amount of logs and summarize them into a small number of event sequences). Since our log linking is designed to be localized, for example, linking *Hadoop* task logs only needs the logs from one task, our approach is parallelizable with minimal effort.

B. Construct Validity

We use abstracted execution logs to perform log clustering and to learn the runtime behaviour. However, the execution logs may not contain all the information of the runtime behaviour. Other types of dynamic information, such as execution tracing, may have more details about the execution of the BDA Apps. We use the execution logs rather than other more detailed dynamic information in this work, because execution logs are readily available and are widely used in practice (leading to no performance overhead). We leverage the logs from the underlying platform of the BDA Apps (e.g., Hadoop) instead of the logs from the Apps themselves. The purpose of this work is not to identify the bugs in the BDA Apps but rather assist in reducing the effort in deploying BDA Apps in a cloud environment. Therefore, the platform logs provide more and better information than application logs.

Identifying the re-occurrences of sub-sequences can also be used in our approach to reduce the event sequences, similar to our method of eliminating repetitions in Section IV. In our experience, we performed sub-sequence detection on the recovered event sequences and found that it did not suppress execution log sequences as good as our repetition elimination approach. In addition, the process of sub-sequence detection is very time consuming and slows down the overall analysis. Therefore, we did not use the sub-sequence detection in practice. For other BDA Apps and other distributed platforms, sub-sequence detection may be effective in reducing the log sequences.

IX. Related Work

In this section, we discuss the related work in two areas.

A. Dynamic Software Understanding

Fischer *et al*. [18] instrument the source code and produce different kinds of visualizations to track and to understand the evolution of software at the module level. Kothari *et al*. [19] propose a technique to evaluate the efficiency of software feature development by studying the evolution of call graphs generated by execution traces. Röthlisberger *et al*. [20] implement an IDE called Hermion, which captures run-time information from an application under development. The run-time information is used to understand the navigation and browsing of source code in an IDE.

Recent work by Beschastnikh *et al*. [21] designed an automated tool that infers execution models from logs. The models can be used by developers to verify and diagnose bugs. Our techniques aim to provide context of logs when deploying BDA Apps in cloud.

In addition, Cornelissen *et al*. [22] perform a systematic survey of using dynamic analysis to assist in program understanding and comprehension. FIELD is a development environment created by Reiss *et al*. [23] that contains the features to dynamically understand the execution of a program. However, the environment is rather designed for traditional application development, and not for the cloud deployment of BDA Apps.

B. Hadoop Log Analysis

Hadoop typically runs on large-scale clusters of machines with hundreds or even thousands of nodes. As a result, large amounts of log data are generated by Hadoop. To collect and analyze the large amounts of log data from Hadoop, Boulon *et al*. built Chukwa [24]. This framework monitors Hadoop clusters in real-time and stores the log data in Hadoop's distributed file system (HDFS). By leveraging Hadoop's infrastructure, Chukwa can scale to thousands of nodes in both collection and analysis. However, Chukwa focuses more on collecting logs without the ability to perform complex analysis.

Tan *et al*. introduced SALSA, an approach to automatically analyze Hadoop logs to construct state-machine views of the platform's execution [25]. The derived state-machines are used to trace the data-flow and control-flow executions. SALSA computes the histograms of the durations of each state and uses these histograms to estimate the Probability Density Functions (PDFs) of the distributions of the durations. SALSA uses

the difference between the PDFs across machines to detect anomalies. Tan *et al*. also compare the duration of a state in a particular node with its past PDF to determine if the duration exceeds a determined threshold and can be flagged as an anomaly.

Another related work to this paper is the approach of Xu *et al*. in [26], which uses the source code to understand the structure of the logs. They create features based on the constant and variable parts of the log messages and apply the Principal Component Analysis (PCA) to detect the abnormal behaviour.

All the above approaches are all designed for system administrators in managing their large clusters. Our approach, on the other hand, aims to assist developers in comparing the deployed system on such large clusters against the development cloud.

X. Conclusion

Developers of BDA Apps typically first develop their application with a small sample of data in a pseudo cloud, then deploy the application in a large scale cloud environment. However, the larger data and more complex environments lead to unexpected executions of the underlying platform. Such unexpected executions and their context cannot be easily uncovered by traditional approaches.

In this paper, we propose an approach to uncover the different behaviour of the underlying platforms for BDA Apps between runs with small testing data and large real-life data in a cloud environment. To evaluate our approach, we perform a case study on Hadoop, a widely used platform, with three BDA Apps. The case study results show the strength of our approach in two aspects:

1) Our approach drastically reduces the verification effort by 86-97% when verifying the deployment of BDA Apps in the cloud.
2) The precision of our approach is comparable with the traditional keyword search approach. However, the problematic logs reported by our approach are much fewer than using keyword search, which makes it possible to manually explore the problematic logs.

In addition, our approach provides additional context information (execution sequences). Based on the context information, developers can explore the execution sequences of the logs to rapidly understand the cause of problematic log lines.

References

[1] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker, "Interactions with big data analytics," *interactions*, vol. 19, no. 3, pp. 50–59, May 2012.

[2] N. Wingfield, "Virtual product, real profits: Players spend on zynga's games, but quality turns some off," *Wall Street Journal*.

[3] "Ebay is powered by hadoop," http://wiki.apache.org/hadoop/PoweredBy.

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

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, Mar. 2007.

[6] S. Sorkin, "Large-scale, unstructured data retreival and analysis using splunk," *Technical paper, Splunk Inc*, 2009.

[7] P. Mundkur, V. Tuulos, and J. Flatow, "Disco: a computing platform for large-scale data analytics," in *Erlang '11: Proc. of the 10th ACM SIGPLAN workshop on Erlang*. New York, NY, USA: ACM, 2011, pp. 84–89.

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

[9] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. Godfrey, M. Nasser, and P. Flora, "An Exploratory Study of the Evolution of Communicated Information about the Execution of Large Software Systems," in *WCRE '11: Proceedings of the 18th Working Conference on Reverse Engineering*, Lero, Limerick, Ireland, October 2011.

[10] 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.

[11] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.

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

[13] "America's most wanted - a metric to detect persistently faulty machines in hadoop," http://hadoopblog.blogspot.com/2010/06/americas-most-wanted-metric-to-detect.html.

[14] D. Cotroneo, S. Orlando, and S. Russo, "Failure classification and analysis of the java virtual machine," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 17–.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD '08: Proc. of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1099–1110.

[16] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of Map-Reduce: the Pig experience," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1414–1425, 2009.

[17] "Amazon ec2," https://aws.amazon.com/ec2/.

[18] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind, "System evolution tracking through execution trace analysis," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 237–246.

[19] 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.

[20] D. Röthlisberger, O. Greevy, and O. Nierstrasz, "Exploiting Runtime Information in the IDE," in *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 63–72.

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

[22] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Softw. Eng.*, vol. 35, pp. 684–702, September 2009.

[23] S. Reiss, *The Field programming environment: A friendly integrated environment for learning and development*. Springer, 1995, vol. 298.

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

[25] 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*. Berkeley, CA, USA: USENIX Association, 2008, pp. 6–6.

[26] 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*. New York, NY, USA: ACM, 2009, pp. 117–132.