POD-Diagnosis: Error Diagnosis of Sporadic Operations on Cloud Applications

[1]Xiwei Xu, [1,2]Liming Zhu, [1,2]Ingo Weber, [1,2]Len Bass, [1]Daniel Sun
[1]SSRG, NICTA
[2]School of Computer Science and Engineering, University of New South Wales
Sydney, Australia
{firstname.lastname}@nicta.com.au

Abstract— **Applications in the cloud are subject to sporadic changes due to operational activities such as upgrade, redeployment, and on-demand scaling. These operations are also subject to interferences from other simultaneous operations. Increasing the dependability of these sporadic operations is non-trivial, particularly since traditional anomaly-detection-based diagnosis techniques are less effective during sporadic operation periods. A wide range of legitimate changes confound anomaly diagnosis and make baseline establishment for "normal" operation difficult. The increasing frequency of these sporadic operations (e.g. due to continuous deployment) is exacerbating the problem. Diagnosing failures during sporadic operations relies heavily on logs, while log analysis challenges stemming from noisy, inconsistent and voluminous logs from multiple sources remain largely unsolved.**

In this paper, we propose *Process Oriented Dependability (POD)-Diagnosis*, an approach that explicitly models these sporadic operations as processes. These models allow us to (i) determine orderly execution of the process, and (ii) use the process context to filter logs, trigger assertion evaluations, visit fault trees and perform on-demand assertion evaluation for online error diagnosis and root cause analysis. We evaluated the approach on rolling upgrade operations in Amazon Web Services (AWS) while performing other simultaneous operations. During our evaluation, we correctly detected all of the 160 injected faults, as well as 46 interferences caused by concurrent operations. We did this with 91.95% precision. Of the correctly detected faults, the accuracy rate of error diagnosis is 96.55%.

Keywords – system administration; cloud; deployment; process mining; error detection; error diagnosis; DevOps

I. INTRODUCTION

Large-scale applications in cloud systems may consist of thousands of nodes with complex software stacks inside each node and dependencies among nodes. Diagnosing operation errors in such applications has always been difficult and there are a large number of works in this field [1]. The recent emergence of DevOps, infrastructure-as-code, and elastic cloud resources are adding some new challenges.

In the past, major sporadic changes (such as upgrade) to large-scale applications would be infrequent and often done during scheduled downtime with careful execution. Now, with high frequency continuous deployment, sporadic changes are being automated by complex scripts (including code that manipulates the infrastructure) and pre-defined triggers, with only occasional human intervention. These sporadic but now *high frequency operations* often touch the entire production system with significant consequences. This has introduced a number of problems. First, fully testing these operation scripts/code is difficult as mimicking the scale and complexity of the real production environment is not easy. Second, traditional exception handling mechanisms are best suited for a single language environment and break down as operations have to deal with different types of error responses from different systems – ranging from the error code of a cloud API call to a potential silent failure of a configuration change. This situation demands robust monitoring, diagnosis and recovery *during* the sporadic operations. Yet, traditional anomaly-detection-based diagnosis approaches (e.g. based on rules, statistics, machine learning) [1] are built for use during "normal" operations. They assume a system has normal operation profiles that can be learned from historical data and deviations from the profiles can help detect, localize, and identify faults.

In this paper, we adopt a model-based approach [1] that explicitly models these sporadic operations as processes and uses the process context to locate errors, filter logs, visit fault trees, and perform on-demand assertion evaluation for online error diagnosis and root cause analysis. In particular, our error diagnosis relies on a *process model* to specify the key steps and the desired system states (through *assertions*) after each key step. We then perform *assertion evaluation* and process *conformance checking* at runtime to locate errors within the process context. We further *diagnose errors* through the use of on-demand assertion evaluations (diagnosis tests) with the help of fault trees (essentially a structured repository of known errors and root causes). In terms of creating the process model, our initial work [2] shows that the process can be discovered using process mining [3] on normally generated logs of successful operations.

A rolling upgrade operation is a prime example of a sporadic operation and we will use it as our case study. We describe rolling upgrade in the next section. Errors during a rolling upgrade must be identified, diagnosed and handled within minutes or seconds as the full production system is at stake. Because of the problems outlined earlier, standard monitoring of normal operations is often deliberately disabled during the rolling upgrade. There is often a long gap between an error and its associated failure and there is no online error diagnosis if the error is detected. The default recovery is usually a complete but equally risky rollback operation (unless expensive redundancy is used).

Error diagnosis during operation time heavily relies on logs. In addition to the unique challenges due to the high frequency and sporadic nature of the operations, there are also log analysis challenges [4] exacerbated by the uncertain cloud environment, multiple distributed log sources and simultaneous confounding operations. Our POD-diagnosis approach also deals with them specifically.

Another challenge to log analysis comes from the voluminous and inconsistent logs from different systems including cloud infrastructure and application ecosystems. Events appearing in the log are often out of the context [4]. Our approach uses Logstash[1] to filter and collect distributed logs from different sources using process/step-specific regular expressions and uses process conformance checking to localize the issues.

The cloud environment introduces uncertainties for operations that have traditionally been under the direct control of an enterprise. Enterprises have limited visibility of a cloud environment and have to rely on cloud infrastructure APIs and technologies to perform their sporadic operations. Many cloud resource-provisioning technologies such as CloudFormation[2] use a declarative black-box approach for achieving the end results. This is equally true for typical configuration management tools like Chef[3] or Cfengine[4], which rely on eventual convergence to achieve desired end states. These tools feature neither online error diagnosis support nor fine-grained targeted healing – only complete rollback/opportunistic retry – if something goes wrong in the middle of the operations. Our approach breaks down a sporadic operation into finer-grained steps, performs conformance checking to detect any deviation from the expected execution of the process, defines intermediate expected outcomes through assertions, defines a fault tree for all the (known) possible intermediate errors and performs diagnosis tests visiting the fault tree for its root causes. Finally, the existing works in error diagnosis lack the awareness of multi-operations and ecosystem context [4]. The built-in exception handling in tools such as Asgard[5] or Chef has only local information about the current operation it handles. Our approach has global visibility as it uses the aggregated and process-annotated logs from different operations in the central repository for error diagnosis.

We implemented our approach in a prototype and evaluated our approach in AWS. The evaluation results show that 95% of our online error diagnosis finishes within 3.83 seconds, with an overall precision of 91.95%, a recall of 100% for error detection, and 97.13% accuracy rate for root cause diagnosis.

Contributions[6]: The key contribution made by our Process-Oriented Dependability (POD)-Diagnosis approach is the use of process context (such as operation process id, instance id, step id, conformance status) to improve the success of error detection and diagnosis in this particular domain. Process context has not been applied before in this domain. Our technique is non-intrusive and does not require any modification to the operation scripts/tools. Also, we use fault trees to capture potential faults and root causes of intermediate errors, and the associated on-demand assertions (diagnosis checks) allow pinpointing root causes.

Another contribution is the evaluation, where we injected faults in 160 runs of a rolling upgrade process on AWS to test accuracy and completeness of POD-Diagnosis. There are no equivalent tools to compare against in this particular domain; hence this is an exploratory study.

The paper proceeds by introducing rolling upgrade in Section II, followed detailed discussions of POD-Diagnosis in Section III. Section IV presents the implementation of POD-Diagnosis. We evaluate our approach in Section V and discuss limitations in Section VI. Related work is discussed in Section VII, and Section VIII concludes the paper.

II. Operations Process Example: Rolling Upgrade

Rolling upgrade is an important element of continuous delivery and high frequency releases. In continuous deployment, a rolling upgrade of the entire system can happen multiple times a day [5] without system downtime.

Assume an application is running in the cloud. It consists of a collection of virtual machine instances, instantiated from a smaller number of different virtual machine images. Say, a new machine image representing a new release for one of the images (VM_R) is available for deployment. The current version of VM_R is V_A and the goal is to replace the N instances currently executing V_A with N instances executing the new version V_B. A further goal is to do this replacement while still providing a sufficient level of service to clients of VM_R: at any point of time during the replacement process, at least N' instances running some version of VM_R should be available for service.

One method for performing this upgrade is called *rolling upgrade* [6]. In a rolling upgrade, a small number of k (with $k=N-N'$) instances at a time currently running version V_A are taken out of service and replaced with k instances running version V_B. By only upgrading k machines, the number of instances in service remains at the desired level of N'. The time taken by the replacement process for one instance is usually in the order of minutes. Consequently performing a rolling upgrade for 100s or 1000s of instances will take on the order of hours. The virtue of rolling upgrade is that it only requires a small number of additional instances to perform the upgrade. Rolling upgrade is the industry standard method for upgrading instances [6].

In this paper, our examples are the provisioning failures that can occur during rolling upgrade. A provisioning failure

[1] Logstash – http://logstash.net
[2] Cloudformation – http://aws.amazon.com/cloudformation/
[3] Chef – http://www.opscode.com/chef/
[4] Cfengine – http://cfengine.com/
[5] Asgard – https://github.com/Netflix/asgard
[6] [2] is an early version of this work. The paper gives an initial idea of providing a holistic process view of system operations. It does not contain conformance checking, error diagnosis, or an in-depth evaluation.

occurs during the replacement process, specifically when one of the upgrade steps produces incorrect results. We do not consider failures due to logical inconsistencies among the versions being active simultaneously.

Current implementations of rolling upgrade vary. We base our discussion of provisioning failures on the method Asgard currently uses. Asgard is Netflix's customized management console on top of AWS infrastructure. For provisioning failures, Asgard will recognize them and report the failure to the operator. The time between the failure occurring and the report to the operator may be as long as 70 minutes. Asgard may not recognize some provisioning failures such as failures caused by concurrent independent upgrades.

III. POD-DIAGNOSIS

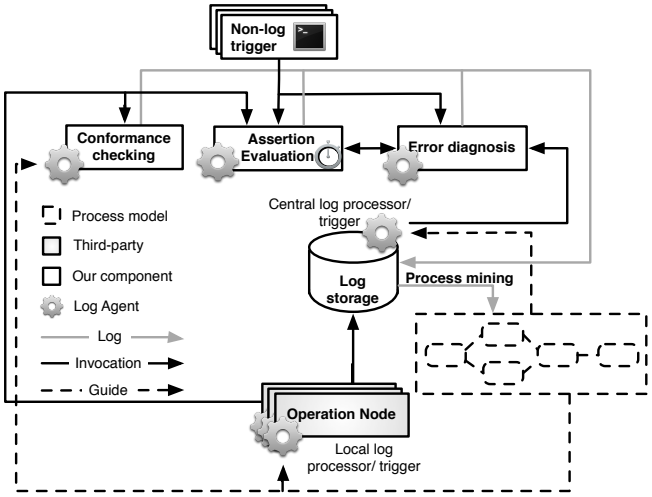

Figure 1. POD-Diagnosis Overview

Figure 1 gives an overview of POD-Diagnosis. The white boxes are the components of POD-Diagnosis; the grey boxes are third party components. A cogwheel on a box represents a log agent. The grey arrows represent log flow; the black arrows represent control flow. We first give an POD-Diagnosis overview, followed by details on the steps.

Logs are our primary sources of information. Some logs are generated by the cloud infrastructure while other logs are generated by an "operation node" (e.g. where Asgard runs) that orchestrates the operation. As we operate in the cloud, some logs are hidden by cloud providers but can be requested through API calls.

The process model is created offline using process mining techniques. The assertions and their evaluation are developed based on the process model, also offline. As the rolling upgrade proceeds, it generates log lines that trigger assertion evaluation and conformance checking. Assertion evaluation is also triggered from non-log sources such as timers and diagnosis. Assertions evaluate the state of the environment to detect anomalies. Detection of an anomaly triggers error diagnosis, which is reported to the operator.

A. *Offline Process Mining and Assertion Creation*

The process model can be created manually or discovered from logs by process mining [3]. The granularity of the model is determined by the analyst. The granularity may be constrained by log granularity if the process is discovered from logs. It is possible to construct a process model from scripts or runbooks and not be constrained by logs. To track the execution of the process, the analyst still needs to specify which log lines belong to which activity.

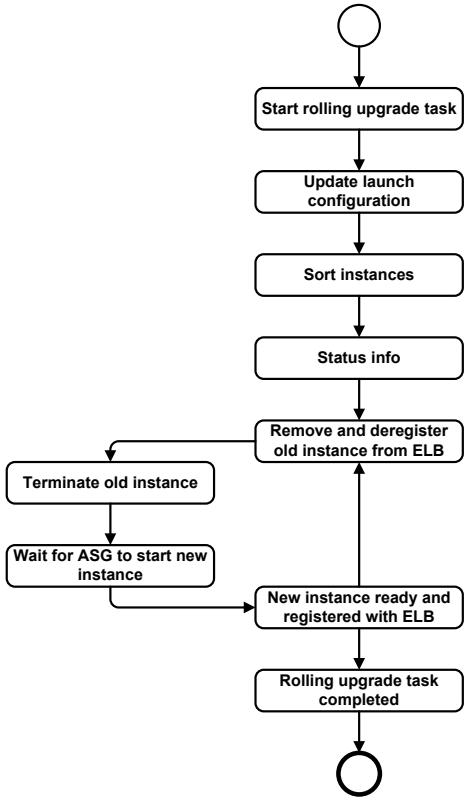

Figure 2. Process model of Rolling Upgrade, including time data.

In our preliminary work [2], we discovered the process model from logs produced by Asgard – Figure 2 shows the resulting process model. For this purpose, we collected the logs from Asgard, clustered the log lines using a string distance metric, and manually combined and named clusters at the desired level of granularity. From this information, i.e., sets of log lines and the corresponding activity names, we derived regular expressions matching the log lines, and formed transformation rules: if ($\texttt{regex}_i$ or $\texttt{regex}_{i+1}$ or …) matches, add tag `[activity name]` to the line. Then we applied these rules to all log lines. The tagged log is provided to an off-the-shelf process mining tool, Disco[7], and resulted

[7] Disco – http://fluxicon.com/disco/

in Figure 2[8]. Discovery is the prime solution in process mining: from a set of event traces (in our case: logs), the algorithms derive causal dependencies between events, e.g., that event A is always followed by event B. By putting all such dependencies together, a process model such as the one shown in Figure. 2, can be derived. However, system logs do not lend themselves naturally as direct input into the tools. Therefore we created the described pre-processing pipeline – as described in detail in [2].

The process model is used to guide the local log processors, central log processor and conformance checking. The local log processor uses the process model to annotate *process context (process ID, instance ID, step ID, outcomes of the steps)* on the log produced at runtime, and trigger other functionalities accordingly. Locations for annotations are typically the beginning or the end of a process step. It is also possible to have annotations during a process step, by making it dependent on a specific type of log line.

For each intermediary step, the expected outcomes can be captured as assertions. Note that the creation of the process model and the assertions needs to be done only once for each process/operation using a particular tool, and are reusable for an arbitrary number of process instances (e.g., concrete rolling upgrades using Asgard). This means that our approach can be used when Asgard is used for a rolling upgrade regardless of the application being upgraded or the number of instances being upgraded.

B. Online Error Detection and Diagnosis

Operations can be performed manually or automatically by an operation agent installed on the application node. Many automated operations need an additional operation node to coordinate.

While the operation is underway, the operation node will produce operation logs. The produced logs are firstly processed by a local log processor agent, which *filters* the original logs, *annotates* the corresponding log lines with process context information, *triggers* post-step assertion evaluation and conformance checking according to the process context, and *forwards* "important" lines to the central log storage. By "important" line, we usually mean the log lines that represent the start or end of a process activity. The failure of assertion evaluation and conformance checking *triggers* Error Diagnosis.

The results of Conformance Checking, Assertion Evaluation and Error Diagnosis are also recorded as logs. They are forwarded to the central log storage and merged with the operation logs collected from distributed nodes. The data in the log storage can be used for future process discovery, e.g. when a process has changed, or offline diagnosis. A central log processor grabs the logs from the central log storage and triggers the error diagnosis when it finds a failure or exception indicated by the log line.

1) Log Processors

[8] The figure has been re-drawn for clarity in print.

Figure. 3 shows the local log processor running on operation node. The local log processor is a pipeline connecting a set of log-processing components. Once the log processor detects a new log line in the operation log file, the log line goes through the processing components within the pipeline.

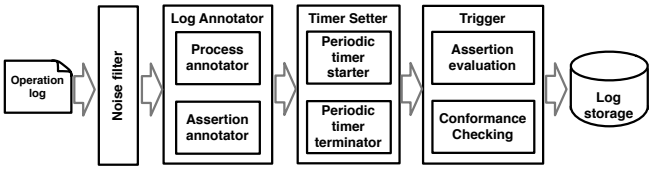

Figure 3. Local Log Processor

Noise filters drop any log line that is not relevant to the current operation process based on regular expressions. The log annotator annotates the matched log line with context information using tags. The timer setter uses the log line indicating the start of the operation process to start the periodic timer and uses the log line indicating the end of the operation process to stop the periodic timer – see Section III.B.3 for details. The trigger uses the matched log line and annotated process context to trigger Conformance Checking and Assertion Evaluation. Finally, relevant log lines are forwarded to the central log storage.

2) Conformance Checking

Conformance checking in process mining refers to methods and algorithms for comparing if an event log fits a process model (or vice versa). Say, the log contains event A followed by B; conformance checking tells us if the process model permits this, or not. This is also referred to as *fitness*: to which degree does the log and the model fit? Several algorithms were proposed to this end in the literature. We make use of the *token replay* technique from [3], Chapter 7.2, adapted from Petri Nets to the semantics of Business Process Model and Notation (BPMN) [7]. Conformance checking can detect the following types of errors:

- *Unknown*: a log line that is completely unknown.
- *Error*: a log line that corresponds to a known error.
- *Unfit*: a log line corresponds to a known activity, but that should not happen given the current execution state of the process instance. This can be due to *skipped activities* (going forward in the process) or *undone activities* (going backwards).

In order to check conformance in a near-real-time fashion, the local log agent sends a message to the service for each event, containing the process model ID (or process ID), the trace ID (or process instance ID), and the whole log line as outputted by respective system.

Upon receiving a message, Conformance Checking looks up the process instance, if it is known; if not, a new instance is created. Then it tries to match the log line against the regular expressions for identifying which process activity this line belongs to. If no match is found, the log line is tagged as `[conformance:unclassified]`. This is commonly the case for error messages or unusual execution paths in the

software, and treated like a detected error. If the line matches a regular expression of a known error, the line is tagged as `[conformance:error]`.

If the activity to which the line belongs was identified, we check if this activity was expected to occur in the current state of the process instance – if it was "activated". If so, we proceed the token replay with the execution of the found activity, store the new process instance state, and tag the log line with `[conformance:fit]`. If the activity was not activated, it was executed out of turn – in which case we tag the line as `[conformance:unfit]`, which also indicates a detected error.

Any detected error triggers error diagnosis (see Section III.B.4). From the current execution state of the process instance, we can derive the process context, except for the outcome of a step – this is done by assertion evaluation. We can further derive the *error context*: the last valid state of the process before the error, the last activity that executed successfully, and the hypothesized skipped/undone activities.

3) Assertion Evaluation

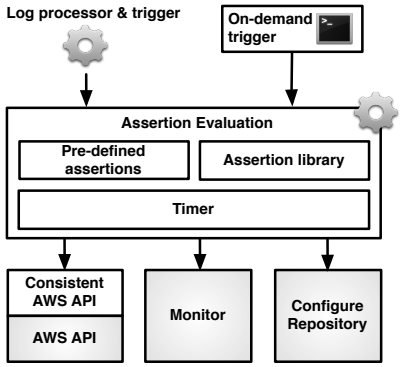

Figure 4. Assertion checking

The expected outcomes of each intermediary step are captured as assertions and evaluated at different times for different purposes (e.g. upon a step's completion for error detection or upon error/failure detection for error diagnosis).

As shown in Figure 4, Assertion Evaluation can be triggered by three mechanisms. The local log processor is the primary trigger, which triggers assertions according to the process context information annotated on the log lines. Other trigger mechanisms include timers and on-demand command triggers. We have two types of timers: one-off timers and periodic timers. A one-off timer is used to check an assertion at a specified time point. Ideally, every step of the operation process produces logs indicating the start, in-progress and the end of the step. Normally, the log line indicating the completion of the step triggers the assertion evaluation. However, sometimes there is no log line indicating the completion of a certain step. In such cases, we set a timer to trigger the corresponding assertion evaluation after a period of time. A periodic timer is used to check an assertion every so often. As we mentioned in Section III.B.1, normally we set the periodic timer at the beginning of an operations process, and stop the periodic timer at the end of the operation process. In cases where a certain log event occurs periodically, we use log to adjust the timer setting to keep the timer aligned with the process context. For example, for a periodic log event we can set a timer when the first log event occurs. The timeout value is set to the duration after which the second log event is expected to occur, plus some slack time. The values are usually based on measured historical timing profiles and process mining. If the second log event occurs before timeout, the corresponding assertion evaluation is triggered and the timer is reset.

Assertion Evaluation uses AWS APIs and third party cloud monitors, such as Edda[9], to check the online status of cloud resources, and configuration repositories to check the configuration values. We added a layer on top of AWS API to deal with the issue of eventual consistency. We provide a set of pre-defined assertions to check cloud resources, which operators can use directly. We also provide an assertion library, which analyst can use to link their assertions with the operation processes.

There are two types of assertions. High-level assertions check the status of the overall system, for example, "assert the system has at least *M* instance with the new version". Low-level assertions check the status of a specific node. The analyst decides whether to use high-level assertions, which takes longer time to diagnose if failure occurs, or use low-level assertions, which introduce more overhead. There are two scenarios in which we use the low-level assertions: (i) to double-check the results if the log indicates acknowledged success of a certain API call, and (ii) to check for subtle errors at the application level or in the configuration.

4) Error Diagnosis

When an assertion evaluation fails, a process non-conformance is detected, or an error/failure is reported by other monitoring systems, Error Diagnosis is triggered to diagnose the causes at runtime.

Error Diagnosis searches for the root causes when a failure occurs. The errors/failures could be caused by the operations process itself, the cloud infrastructure, co-located applications or other simultaneous operations. The contribution of our diagnosis comes largely from on-demand assertions defined for each type of error/failure. Other diagnostic information can come from third party monitors or configuration repositories, which may provide data on who changed the configuration, when, and why.

Since our POD-Diagnosis is oriented towards Cloud operations, the root causes consequently refer to those events in operation levels, such as exceptions, misconfigurations, system call fails, etc. From the root causes to the potential errors detected by the error detection, there are a large number of intermediate events that connect root causes to errors. For an efficient diagnosis, all the events and the dependencies among them should be well structured. Further, the events including possible failures/errors, their associated potential faults, and on-demand assertions can be naturally organized into tree-like structures. Thus, we created fault trees to serve as a reference model for both robust operations

[9] Edda – https://github.com/Netflix/edda

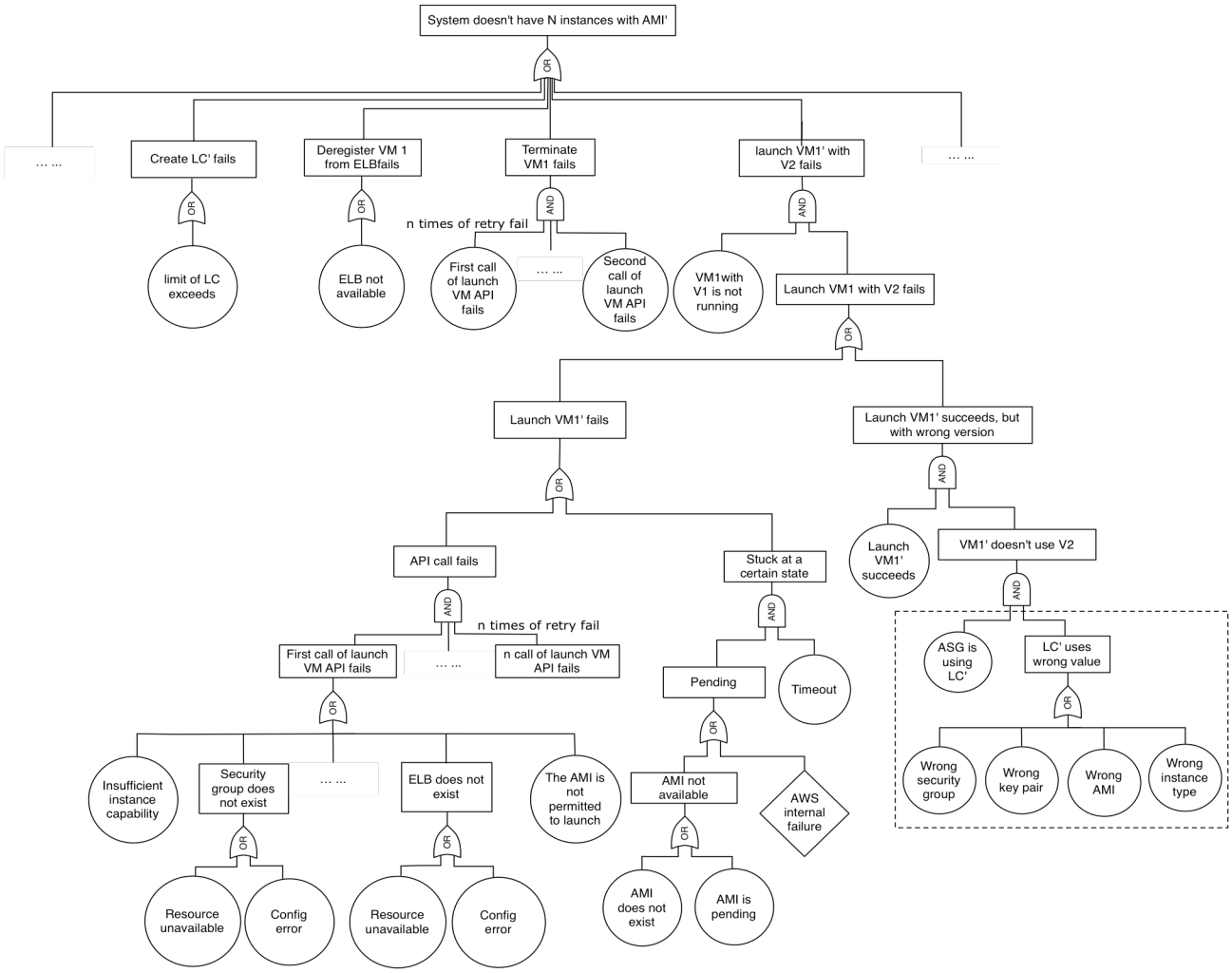

Figure 5. Part of a Fault Tree

design and error diagnosis. In contrast to traditional fault tree analysis (FTA) for hardware architectures, the fault trees here are constructed from and based on application system functions and knowledge of their possible faults. Note that the fault trees are not employed for FTA; instead we use them to structure data in a repository.

Figure 5 shows part of a fault tree derived from the failure of assertion "assert the system has *N* instances with the new version". There is one fault tree per assertion. As can be seen in the figure, there are some variables in the tree. When the Error Diagnosis is triggered, we firstly select the correct tree(s) according to the assertion that triggered the diagnosis. Secondly we instantiate the variables in these trees with the parameters from the runtime request, e.g. the number of nodes (*N*). Then the associated process context from the request is used to prune sub-trees that are not relevant in that process context. For example, in Figure 5, the left-most sub-tree (below *Create LC' fails*) is associated with the step *Update launch configuration* (see Figure 2) as process context, and the right-most sub-tree with step *New instance ready…* . If the assertion after *New instance ready…* triggered diagnosis, we prune all other sub-trees.

After selecting and instantiating the context-specific sub-trees, the corresponding diagnostic tests (on-demand assertion evaluation / consulting monitoring/repository) are determined by visiting these sub-trees in a top-down manner, so as to confirm or exclude potential faults. If the check at a particular node has already been done, e.g. for an ancestor node, the diagnosis results are reused. The diagnostic tests that are actually performed are highly dynamic and compositional depending on the current situation. If the test for a current node shows the related error is not present, we proceed elsewhere (and prune the respective sub-tree); if the error is present, we visit the child nodes. Diagnosis stops at the point where no further child nodes can be checked, e.g. when an instance was terminated, but the diagnosis cannot determine why. Any faults from descendant nodes below the deepest successful error tests *might* be the root causes. The partial log below shows an example result from diagnosis. The corresponding sub-tree is highlighted by the dashed box

in Figure 5. In this particular run, the assertion that a new instance uses the correct version failed, because the launched instance is based on the wrong AMI.

```
...
[2013-11-19 11:48:01,100] [diagnosis] [ProcessInsId] [stepId] Performing on demand assertion checking: The ASG ASG-dsn is using a correct version. 4 potential faults in total...
[2013-11-19 11:48:01,101] [diagnosis] [ProcessInsId] [stepId] [Assertionid] Verifying the security group setting of the ASG ASG-dsn
[2013-11-19 11:48:01,187] [diagnosis] [ProcessInsId] [stepId] [Assertionid] Verified the security group setting: The ASG ASG-dsn is using a correct security group. 1/4 fault is excluded
[2013-11-19 11:48:01,188] [diagnosis] [ProcessInsId] [stepId] [Assertionid] Verifying the key pair setting of the ASG ASG-dsn
[2013-11-19 11:48:01,262] [diagnosis] [ProcessInsId] [stepId] [Assertionid] Verified the key pair setting: The ASG ASG-dsn is using a correct key pair. 2/4 faults are excluded
[2013-11-19 11:48:01,263] [diagnosis] [ProcessInsId] [stepId] [Assertionid] Verifying the AMI setting of the ASG ASG-dsn
[2013-11-19 11:48:01,333] [diagnosis] [ProcessInsId] [stepId] [Assertionid] Failed verification of the AMI setting: The ASG ASG-dsn is using a wrong AMI.
[2013-11-19 11:48:01,333] [diagnosis] [ProcessInsId] [stepId] One root cause is identified
...
```

At this stage, the order in which potential faults are examined is determined by the fault probability. Another option would be to consider the expected time/cost of the diagnostic tests when determining the test order.

C. Discussion of Effort

The effort on model discovery, log annotation configuration, assertion specification and fault tree creation only needs to be spent only once for an operation tool, such as Asgard. The effort could thus be spent once by the vendor / open-source community. An individual organization using Asgard to perform their rolling upgrade does not need any additional effort to benefit from our system. In particular, no effort is required per rolling upgrade execution.

Even though the effort is once-off, the process model creation part (and associated regular expressions for triggering assertions) is highly automated using process mining techniques [2]. The effort should be in the order of a few hours or less, given suitable logs are readily available. Evolution of the process model can be achieved by amending the set of training data with more logs as they become available. As such, the model can be extended or changed to fit the amended data. Model building is essentially an orthogonal issue and this paper demonstrated how having such a model improves dependability.

Although the implementation (discussed Section IV) is specific to the rolling upgrade process provided by Netflix Asgard, the fault trees form a valuable knowledge base reusable in any sporadic operations using the cloud API and in other diagnosis situation (sporadic or not). Conformance Checking is purely automatic, given the process model. This shows the approach is generalizable to other operations.

IV. Implementation

Our log processors are based on Logstash – an open source tool for managing events and logs. The log processors use regular expressions to match log lines, and process the matched log lines. The code snippet below gives an example of the Logstash entries of a log line from the original operation process with our process context tags – note that `@message` contains the original log line.

```
{"@source":"asgard.log","@tags":["push","asg","step4"],
"@fields":{"time":["2013-10-24 11:41:48, 312"],
"amiid" :["ami-750c9e4f"], "asgid":["pm--asg"],
"appid":["pm"], "instanceid" :["i-7df34041"],
"num":["4"]},"@timestamp":"2013-10 24T00:41:48.855Z",
"@source_host":"NICTA.local","@source_path":"asgard.log
","@message":"[2013-10-24 11:41:48,312] [Task:Pushing
ami-750c9e4f into group pm--asg for app pm]
com.netflix.asgard.Task 2013-10-24_11:41:48 908:
{Ticket: null} {User: null} {Client: localhost
0:0:0:0:0:0:0:1%0} {Region: ap-southeast-2} [Pushing
ami-750c9e4f into group pm--asg for app pm] Instance pm
on i-7df34041 is ready for use. 4 of 4 instance
relaunches done.","@type":"asgard"}
```

Conformance Checking, Assertion Evaluation and Error Diagnosis are implemented as RESTful Web Services, based on RESTlet[10] — a RESTful Web API framework for Java. The process model is provided to the services up-front. At the moment, assertion evaluations are implemented in a combination of Java and scripts, depending on what is to be asserted. The pre-defined assertions are developed by us, specific to Asgard, and are based on our understanding.

The log line below gives an example of the assertion result produced by the assertion evaluation, which was triggered by the log line shown above.

```
{"@source":"assertion-evaluation.log","@tags":["pushing
pm--asg","step4"],"@fields":{"time": ["2013-10-24 11:
42:00,319"],"taskid":[" pushing pm--asg"],"steppostcon
":["step4"]},"@timestamp":"2013-10-24T00:42:00.319Z",
"@source_host":"NICTA.local","@source_path":"postcondit
ion.log","@message":"[2013-10-24 11:42:00,319]
[assertion] [Task:pushing pm--asg] [Step:step4] ASG pm-
-asg has 4 instances. ","@type":"assertion"}
```

To be resilient against AWS API inconsistency [8], we also implemented a consistent AWS API layer. This includes an exponential retry mechanism: if the supposed status of a specific cloud resource is different from our expectation we retry the respective AWS API calls automatically. We also introduce an API timeout mechanism: assertion evaluations are regarded as failed if API calls time out. Timeout values are set based on experiments, at the 95% percentile.

V. Evaluation

In order to evaluate the POD-Diagnosis approach, we injected real faults in a realistic environment. In this section, we report the diagnosis time, precision and recall of error detection, and accuracy of root cause diagnosis. The next section discusses the results and some general limitations.

[10] RESTlet – http://restlet.org

A. Evaluation Methodology

As our approach is specifically designed for sporadic operations (rather than normal operations), it would be unfair to compare it with diagnosis approaches designed for normal operations. The current practice in industry is to disable most systems for monitoring, automated log analysis, and automated diagnosis during a sporadic operation. If something goes wrong, the sporadic operation is rolled back and an operator conducts a manual diagnosis by running tools from a command line interface and checking various logs. To some degree, POD-Diagnosis automates this manual diagnosis. Our diagnosis time is within seconds, so it would also not be appropriate to compare with manual diagnosis, which may take minutes at least.

Our experiment is running on AWS. We performed rolling upgrade on clusters with 4 or 20 instances. Netflix Asgard is chosen to assist the rolling upgrades. We injected 8 different types of faults into the clusters. Those faults are reported by industry outage reports or experienced by us during rolling upgrade. These faults are representative.

For each of the 8 faults, we conducted 20 runs (i.e., 160 runs in total). We also injected simultaneous operations (such as legitimate scaling in/out or changes to instances) to confound our diagnosis. Among the 20 runs, we mixed a combination of different simultaneous operations. The operations were run in an AWS account shared with an external independent research team whose use may also introduce some confounding factors, such as desired instance number, launch configuration, and call limits imposed on a specific region of a single account.

B. Experiment Setup

The application on which the rolling upgrade is performed is a distributed log monitoring system consisting of Redis[11], Logstash, ElasticSearch[12], and Kibana[13], running on the Ubuntu operating system. Each deployed instance of the application can be used to aggregate distributed logs produced by the customer's own applications.

We have a cluster with 4 or 20 instances deployed in AWS as an ASG. The ASG-based cluster is associated with an Elastic Load Balancer (ELB) to provide a point of contact for incoming traffic. When the ASG finds an unhealthy instance (e.g. caused by termination), it replaces the instance with a new one. Asgard does rolling upgrade by terminating instances from the ASG, and utilizing the ASG to launch new instances with the new version – see Figure 2.

The instances within the ASG are associated with one Security Group (SG). All the instances use the same Key pair for connection. Upgrading an instance could include changing AMI, SG, Key or other configurations, such as instance type, kernel ID, or RAM disk ID. In our experiment, we upgraded 1/4 nodes at a time if the cluster size was 4/20, respectively, during rolling upgrade.

[11] Redis – http://redis.io/
[12] ElasticSearch – http://www.elasticsearch.org/
[13] Kibana – http://rashidkpc.github.io/Kibana/

Our experiment uses a combination of step-specific assertions and high-level overall assertions at different stages of the rolling upgrade. For example, we "assert the system has N instances with the new version" after each completion of the loop, as mentioned in Section III.B.4. In parallel to assertion evaluation, each log line is forwarded to the conformance checking service.

To simulate a complex ecosystem, we ran another small simultaneous operation in parallel to rolling upgrade – ASG's scaling-in. We also randomly terminated instances to increase the uncertainty of cloud infrastructure. Our approach did detect such errors, but could not diagnose the root causes without information like which AWS API calls happened – see the discussion on CloudTrail in Section VII. For example, in our experiment we were able to diagnose when the root cause was ASG scale-in, but not when the root cause was termination of instances.

C. Fault Injection

We ran our experiment by injecting one fault into the process at a random point of time during rolling upgrade. POD-Diagnosis can tackle three types of faults at this stage: (i) faults causing mixed versions due to unexpected simultaneous upgrades, (ii) resource faults, and (iii) configuration faults. We injected the following 8 representative faults in our experiment:

1. AMI changed during upgrade
2. Key pair management fault
3. Security group configuration fault
4. Instance type changed during upgrade
5. AMI is unavailable during upgrade
6. Key pair is unavailable during upgrade
7. Security group is unavailable during upgrade
8. ELB is unavailable during upgrade

One of the most challenging faults is the ASG mixed version error, which can be caused by two simultaneous rolling upgrades. In a large-scale deployment, this can happen quite easily if different development teams push out changes independently during a relatively long upgrade process. This can result in mixed versions of the system co-existing. Some faults, such as AMI changes, essentially simulate the continuous deployment scenario where largely independent teams push small updates to the production environment through rolling upgrade. There might be conflicts or race conditions during these rolling upgrades.

Some faults, such as key pair management, correspond to classical operator problems around ssh-based command orchestration and security. Other faults, such as security group settings, represent the configuration errors, the complexity of infrastructure and networks settings. A third class of faults, such as Elastic Load Balancer (ELB) faults and resource unavailability represent the uncertainty of the infrastructure. For example, an AWS ELB service disruption [9] was caused by "missing ELB state data" last year.

D. Experiment Results

Conformance checking was used to detect errors and derive the error context. The first 4 fault types are not detectable by conformance checking (since the log output is the same). Out of the remaining 4 fault types with 20 runs each (i.e., 80 runs in total), conformance checking found that 20 produced erroneous log traces *before* assertion checking. When called locally, the conformance checking service responded on average in about 10ms.

TABLE I. EVALUATION METRICS

	Detected	**Undetected**
Injected fault	TP_{det}	FN_{det}
No fault	FP_{det}	TN_{det}
Precision of Detection	$P_{det} = \frac{TP_{det}}{TP_{det} + FP_{det}}$	
Recall of Detection	$R_{det} = \frac{TP_{det}}{TP_{det} + FN_{det}}$	
Accuracy Rate of Diagnosis	$AR = \frac{Num_{correct}}{TP_{det} + FP_{det}}$	

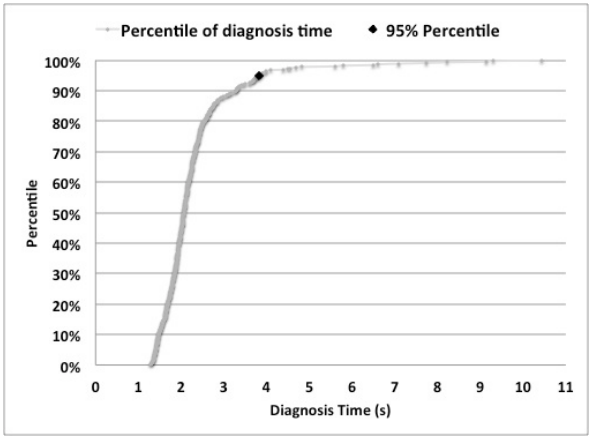

Figure 6. Distribution of error diagnosis time

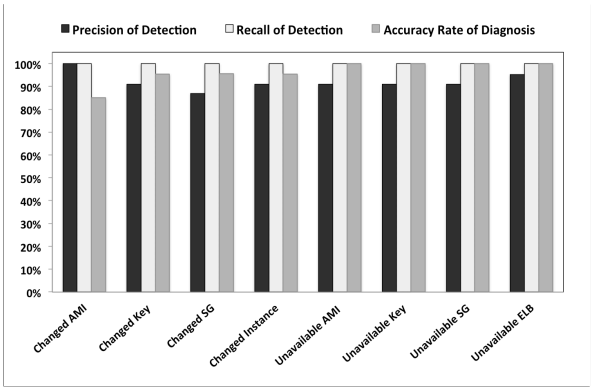

Figure 7. Precision/Recall of Detection/Accuracy Rate of Diagnosis

The evaluation shows that 95% of the online error diagnosis finished within 3.83 seconds. The overall precision was 91.95%, the recall was 100% for error detection, and the accuracy rate for root cause diagnosis was 97.13%. Table I shows the formulas we used to calculate these percentages, where T,F,P,N mean true/false/positive/negative – e.g. TP is the number of true positives. Of the correctly detected faults, we calculate the accuracy rate of diagnosis by dividing the number of root causes being accurately diagnosed by the number of errors being detected. In the cases of FP_{det}, the accurate diagnosis tells us "No root cause identified".

In Figure 6, we give the distribution of error diagnosis time of the 160 runs. The range of the diagnosis time is from 1.29s to 10.44s. The average diagnosis time is 2.30s.

The values of detection precision, detection recall and diagnosis accuracy rate grouped by fault type are shown in the column chart of Figure 7.

VI. DISCUSSION AND LIMITATIONS

A. Discussion

All 160 faults we injected in our experiment were detected by our assertion evaluation. Thus, FN_{det} of all the 8 faults are 0, and recall of error detection is 100%. The assertion evaluations triggered by log provide an effective detection. The reasons for this high rate are discussed below.

Due to some limitations of our approach, there are false positives of detection, and wrong diagnosis. We give a few examples here and discuss the limitation in the next section.

One class of FP_{det} is caused by the error detection triggered due to timeout. The timeout setting comes from historical data. There are rare cases that an operation is running successfully, with late log appearance, which causes the assertion evaluation to fail. However, in all such cases, our diagnosis returned "No root cause identified". This issue could be solved by increasing the timeout values, at the cost of later detection of actual errors.

The second class of FP_{det} is caused by an unexpected long assertion evaluation. In such cases, when the assertion evaluation asserts the number of instances, the "should-be" number is changed by another assertion evaluation thread, which caused the assertion evaluation to fail.

One class of wrong diagnosis comes from purely timer based assertion evaluation failure. As it was not triggered by log lines, there was very limited information (e.g. no instance id) on the target that diagnosis tests can check.

The second class of wrong diagnosis comes from multiple diagnoses across rolling upgrade loops. A changed AMI fault happened and it caused some instances of the wrong version to be launched. Multiple assertion failures and associated diagnosis tests are conducted. During this period, the AMI changed again causing the diagnosis results to be different across different diagnosis tests.

The third class of wrong diagnosis comes from a transient launch configuration change fault. The fault injection mechanism injected a fault and then corrected it soon after. When the on-demand diagnosis test tried to confirm the root cause, it could not find any faults since the root cause had effectively been removed by that time. To mitigate this, one could consult other sources of information, such as CloudTrail – which can take fairly long to provide the relevant information.

The fourth class of wrong diagnosis comes from interference by the other independent team sharing the same AWS account. The simultaneous operation on a different set of instances caused the account instance limit to be reached. Our fault tree did not capture this as a potential root cause, thus the diagnosis stopped at an error on exceeding limit. We amended the on demand assertions and the root cause so that we can correctly diagnose this fault in the future.

Some assertions are added because of the subtle errors or potential causes we learn about over time (in addition to the assertions simply capturing expected intermediate outcomes). They act *like regression tests*. That is both a strength and a weakness. The strength is shown by the high rate of detection of injected errors. The weakness is that previously unseen errors may pass undetected. However, the assertions about expected intermediate outcomes can already capture many important errors. On the other hand, conformance checking finds deviations from the usual log behavior, and does not require manual specification of known errors. However, conformance checking only detects errors that can be detected from changed log behavior. These two types of triggering are complementary. The ability of conformance checking to detect a high number of errors in the absence of assertions is part of an ongoing investigation.

B. Limitations

First, our approach is designed for sporadic operations. As mentioned earlier, due to the emerging "continuous" high frequency nature of sporadic operations, the approach could be deployed alongside normal operations. It is not meant to replace normal operation monitoring and error diagnosis.

Second, our approach at the moment heavily relies on logs, whose quality varies across different software and cloud infrastructure providers. If there is very limited information in logs, the process discovery, conformance checking, and assertion-based error detection might be severely affected. Because logs are extremely important in system management and troubleshooting, in contemporary production systems integrated logs and logging sub-systems have become a MUST. Moreover, administrators and engineers strongly desire well-formatted and easily understandable logs and autonomous system monitors request machine-readable logs. Thus, our requirements on logs are becoming more realistic through the evolution of the industry. Our assertion evaluation also supports timers and can be triggered by non-log sources, which overcomes the limitation to a degree. But as discussed above, inappropriate timeout settings can introduce more false positives.

Third, for online diagnosis, the information staleness and timeliness in various monitoring facilities is essential. For example, we could not use the newly released CloudTrail for API call logs due to the major delay between a call and the associated log appearing. On the other hand, the fault might be transient and may have disappeared when the on-demand diagnosis tests are conducted.

We note there are limitations that are intrinsic to our approach such as relying on logs and targeting sporadic operations. Other limitations are due to the quality of the current set of assertions, fault trees and diagnosis tests. One can always improve them to detect unforeseen ones.

VII. Related Work

A. Error Handling in Operation Processes

When an operation process is automated through scripts and the infrastructure-as-code approach, the error handling mechanisms within the scripting or high-level languages can detect and react to errors through exception handling, for example error handlers in Asgard, AWS API error code, such as [10], and Chef's fault handlers [11]. These exception handing mechanisms are best suited for a single language environment breaking down, as operations have to deal with different types of error responses from different systems. Exception handling also only has local information rather than global visibility when an exception is caught. That's why run time external monitoring, diagnosis and recovery are needed.

Most of the configuration management (e.g. CFEngine) and cloud resource provisioning mechanisms (e.g. CloudFormation) use a convergence-based or declarative approach where they keep retrying and waiting or exit if something is wrong. As the execution order is non prescriptive or behind a black box engine, there is often very little can be done on diagnosing intermediate errors.

B. Configuration Error Detection and Diagnosis

As traditional operations are often about changing configuration, there are a large number of configuration error detection and diagnosis tools and research.

Confdiagnoser [12] uses static analysis, dynamic profiling and statistical analysis to link the undesired behavior to specific configuration options. It could diagnose both crashing and non-crashing configuration errors. The technique requires instrumentation around the configuration options in source code and does not diagnose the root cause. ConfAid [13] uses information-flow tracking to analyze the dependencies between the error symptoms and the configuration entries to identify root causes. ConfAid uses dynamic taint analysis to diagnose configuration problems by monitoring causality within the program binary as it executes. The approach is largely for diagnosing a single program's configuration rather than configuration errors introduced during operation. CODE [14] is a tool automatically detects software configuration errors, which is based on identifying invariant configuration access rules that predict what access events follow what contexts. It requires no source code, application-specific semantics, or heavyweight program analysis. Using these rules, CODE can sift through a voluminous number of events and detect deviant program execution. Our approach is complimentary to CODE by detecting a wider source of errors and diagnosing them.

C. Log analysis tools

There are a large number of log analysis tools that can be used for error diagnosis purposes. First, all of them do not conduct online diagnosis through automatically performing diagnosis tests. Second, there is no process context used

during diagnosis. The logs are usually organized around sources and only provide manual drilling down.

In Nov. 2013, AWS announced a new product called CloudTrail[14]that logs all API calls. We evaluated the product, but the delay (up to 15 minutes) between a call and its CloudTrail log appearing is not suitable for online diagnosis.

SEC [15] is an event correlation tool for advanced event processing, which can be harnessed for event log monitoring, for network and security management, for fraud detection, and for any other task, which involves event correlation. Event correlation is a procedure where a stream of events is processed, in order to detect (and act on) certain event groups that occur within predefined time windows. SEC reads lines from files, named pipes, or standard input, matches the lines with patterns (like regular expressions or Perl subroutines) for recognizing input events, and correlates events according to the rules in its configuration file(s). SEC can produce output by executing external programs (e.g., snmptrap or mail), by writing to files, by sending data to TCP and UDP based servers, by calling precompiled Perl subroutines, etc. LogMaster [16] is an event correlation mining system and an event prediction system. LogMaster parses logs into event sequences where each event is represented as an informative nine-tuple. The correlations between non-failure and failure events are very important in predicting failures.

Source code is the schema of logs. [17] parses console logs by combining source code analyses. Then it applies machine learning techniques to learn common patterns from a large amount of console logs, and detect abnormal log patterns that violate the common patterns. Sherlog [18] is proposed as a tool to analyze source code by leveraging information provided by run-times log to infer what must or may have happened during the failed production run. It requires neither re-execution of the program nor knowledge on the log's semantics. It could infer both control and data value information regarding to the failed execution. It uses runtime log to narrow down the possibilities in terms of both execution paths and states during the failed execution in the source code.

D. Intrusive Log improvement for error diagnosis

While there are a number of "rules of thumb" [19] for designing better logging messages, these still do not capture the specific information. LogEnhancer [20] automatically enhances existing logging code to aid future post-failure debugging. LogEnhancer modifies each log message in a given piece of software to collect additional causally related information to ease diagnosis in case of failures. It is the first attempt to systematically and automatically enhance log messages to collect causally-related information for diagnosis in case of failures. [21] proposes an approach based on software fault injection to assess the effectiveness of logs to keep track of software faults triggered in the field. It provides an approach to assess the built-in log effectiveness in a quantitative way. This work motivates their follow-up work [22], which leverage artifacts produced at system design time and puts forth a set of rules to formalize the placement of the logging instructions within the software fault source code to make logs effective to analyze software failures.

[14] Cloudtrail – http://aws.amazon.com/cloudtrail/

E. Error Diagnosis during Normal Operation

There are also statistics, machine learning and rule-based approaches for diagnosing errors during normal operation [23-27]. As noted earlier, they are built for use during "normal" operations. They assume a system has normal operation profiles that can be learned from historical data and deviations from the profiles can help detect, localize and identify faults.

Another category of tools are distributed tracing tools such as Google's Dapper [28], Twitter's Zipkin [29] and Berkeley's X-Trace [30]. They are usually used manually during diagnosis. There is also no notion of processes and process contexts.

VIII. Conclusion and Future Work

Error diagnosis *during* sporadic operations is difficult, as the systems are constantly changing and there are no "normal" profiles to help to detect anomalies. In the past, such sporadic operations were less frequent and done at off-peak times or during scheduled down time. But nowadays, continuous deployment practices are turning sporadic operations into relatively high-frequency occurrences. In this paper, we propose POD-Diagnosis which treats a sporadic operation as a process, detects and diagnoses intermediate step errors through conformance checking, assertion evaluation, and on demand diagnosis tests. The evaluation results are promising, with percentages above 90% for precision, recall, and diagnosis accuracy. We also discussed weaknesses of our approach, including types of errors that are challenging to detect with POD-Diagnosis.

In order to simplify specifying boilerplate assertions, we are designing an assertion specification language at the moment. We plan to automate the generation of assertions. We are also expanding the scope into more types of configurations and operation tasks inside a virtual machine. Finally, we are working towards releasing most of the tools and services described as open-source software.

Acknowledgements

We would like to thank all the reviewers, especially Matti Hiltunen, for guiding us to improve the paper. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] S. Kavulya, K. Joshi, F. Giandomenico, and P. Narasimhan, "Failure Diagnosis of Complex Systems," *Resilience Assessment and Evaluation of Computing Systems*, K. Wolter, A. Avritzer, M. Vieira and A. van Moorsel, eds., pp. 239-261: Springer Berlin Heidelberg, 2012.

[2] X. Xu, I. Weber, L. Bass, L. Zhu, H. Wada, and F. Teng, “Detecting Cloud Provisioning Errors Using an Annotated Process Model,” in The 8th Workshop for Next Generation Internet Computing (MW4NG2013), Beijing, China, 2013.

[3] W. v. d. Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*: Springer Verlag, 2011.

[4] A. Oliner, A. Ganapathi, and W. Xu, “Advances and challenges in log analysis,” *Communications of the ACM*, no. 55, pp. 2, 2012.

[5] Etsy, "Infrastructure upgrades with Chef," *Code as Craft*, 2013.

[6] T. Dumitra¸s, and P. Narasimhan, “Why Do Upgrades Fail and What Can We Do about It? Toward Dependable, Online Upgrades in Enterprise System,” in ACM/IFIP/USENIX 10th International Middleware Conference (MIDDLEWARE2009), Urbana Champaign, Illinois, USA, 2009.

[7] OMG, "Business Process Model and Notation V2.0," 2011.

[8] E. Martin, "Dealing with Eventual Consistency in the AWS EC2 API," *Cloud Foundry Blog*, 2013.

[9] AWS. "Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region," http://aws.amazon.com/message/680587/.

[10] AWS. "Error Codes--Amazon Elastic Compute Cloud," http://docs.aws.amazon.com/AWSEC2/latest/APIReference/api-error-codes.html.

[11] OpsCode. "About Exception and Report Handlers," http://docs.opscode.com/essentials_handlers.html.

[12] S. Zhang, and M. D. Ernst, “Automated Diagnosis of Software Configuration Errors,” in 35th International Conference on Software Engineering (ICSE 2013), San Francisco, CA, USA, 2013.

[13] M. Attariyan, and J. Flinn, “Automating configuration troubleshooting with dynamic information flow analysis,” in The 9th USENIX Conference on Operating Systems Design and Implementation (OSDI2010), Vancouver, BC, Canada, 2010.

[14] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, “Context-based Online Configuration-Error Detection,” in the 2011 USENIX conference on USENIX annual technical conference (USENIX ATC2011), Portland, OR, 2011.

[15] J. P. Rouillard, “Real-time log file analysis using the Simple Event Correlator (SEC),” in The 18th Large Installation System Administration Conference (LISA2004), Atlanta, GA, 2004.

[16] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, and G. Lu, “LogMaster: Mining Event Correlations in Logs of Large-scale Cluster Systems,” in The 31st International Symposium on Reliable Distributed Systems (SRDS2012), Irvine, California, 2012.

[17] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting Large-Scale System Problems by Mining Console Logs,” in The 22nd ACM Symposium on Operating Systems Principles (SOSP2009), Big Sky, MT, 2009.

[18] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, “SherLog: Error Diagnosis by Connecting Clues from Run-time Logs,” in The 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS2010), Pittsburgh, Pennsylvania, US, 2010.

[19] S. Schmidt, "7 More Good Tips on Logging," 2009.

[20] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, “Improving Software Diagnosability via Log Enhancement,” in The 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS2011), Newport Beach, California, USA, 2011.

[21] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia, “Assessing and improving the effectiveness of logs for the analysis of software faults,” in The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010), Chicago, IIIinois, USA, 2010.

[22] M. Cinque, D. Cotroneo, and A. Pecchia, “Event Logs for the Analysis of Software Failures: A Rule-Based Approach,” *IEEE Transactions on Software Engineering,* vol. 39, no. 6, 2013.

[23] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "CloudPD: Problem determination and diagnosis in shared dynamic clouds." pp. 1-12.

[24] P. Xinghao, T. Jiaqi, K. Soila, G. Rajeev, and N. Priya, “Ganesha: blackBox diagnosis of MapReduce systems,” *SIGMETRICS Perform. Eval. Rev.,* vol. 37, no. 3, pp. 8-13, 2010.

[25] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "FChain: Toward Black-box Online Fault Localization for Cloud Systems ".

[26] T. Jiaqi, P. Xinghao, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan, "Kahuna: Problem diagnosis for Mapreduce-based cloud computing environments." pp. 112-119.

[27] P. K. Soila, D. Scott, J. Kaustubh, H. Matti, G. Rajeev, and N. Priya, “Draco: Statistical diagnosis of chronic problems in large distributed systems,” in Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2012.

[28] B. H. Sigelman, and Luiz André Barroso, *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*, Google, 2010.

[29] Twitter. "Distributed Systems Tracing with Zipkin," https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin.

[30] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, “X-Trace: A Pervasive Network Tracing Framework,” in The 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07), Cambridge, MA, 2007