Crying Wolf and Meaning it: Reducing False Alarms in Monitoring of Sporadic Operations through POD-Monitor

Xiwei Xu, Liming Zhu, Min Fu, Daniel Sun, An Binh Tran, Paul Rimba, Srinir Dwarakanathan, Len Bass
SSRG, NICTA
School of Computer Science and Engineering, UNSW
Sydney, Australia
firstname.lastname@nicta.com.au

Abstract—When monitoring complex applications in cloud systems, a difficult problem for operators is receiving false positive alarms. This becomes worse when the system is sporadically being changed and upgraded due to the emerging continuous deployment practice. Other legitimate but sporadic maintenance operations, such as log compression, garbage collection and data reconstruction in distributed systems can also trigger false alarms. Consequently, traditional baseline-based anomaly detection and monitoring is less effective. A normal but dangerous practice is to turn off normal monitoring during sporadic operations such as upgrade and maintenance. In this paper, we report on the use of the process context information of sporadic operations to suppress false positive alarms. We use the context information both directly and in machine learning. Our experimental evaluation shows that 1) using process context directly improves the alarm precision up to 0.226 (36.1% improvement); 2) using process-context trained machine learning models improves the precision rate up to 0.421 (84.7% improvement).

Index Terms—Alarm, Monitoring, Operation

I. INTRODUCTION

In Aesop's story The Boy Who Cried Wolf, the boy falsely crying "wolf" caused the townspeople to ignore his true alarm when a real wolf came. Fast-forward several thousand years.

Modern large-scale applications in cloud systems may consist of thousands of nodes with complex software stacks inside each node and dependencies among nodes. The associated monitoring system has many metrics covering different aspects of the application. A consequence of involving many metrics is that detecting a true anomaly in some metrics and triggering alarms can pose very tricky trade-offs. A problem for operators is to receive false positive alarms or a flood of alarms from different channels about the same event. Under such conditions, operators will quickly get “alert fatigue” and start ignoring alerts or simply turn some of them off. On the other hand, if the operator tries to reduce false positives, s/he may risk missing important events, increasing false negatives. If the alarm settings have very specific assumptions and expectations about a system’s operating condition, the operators may be alerted about some subtle errors. However, the operators may risk rendering the alarms less effective when the system undergoes changes or encounters legitimate but previously unknown external conditions. A large-scale system often needs to perform sporadic maintenance operations such as log compression, garbage collection and data reconstruction in distributed systems. They inevitably impact resource utilization such as CPU usage or network in/out. These legitimate operations may trigger alerts. The recent emergence of continuous delivery and cloud elasticity is adding new challenges to this.

One challenge comes from constant changes. An anomaly is usually signaled by a deviation from normal behavior. Normal behavior assumes the system is relatively stable over time. However, in a large-scale complex application being continuously delivered/upgraded, changes are the norm. We are not talking about varying workloads or dynamic aspects of the monitored applications, which are often well anticipated. The new challenges come from the continuous delivery/deployment operations themselves. For example, an automated system operation task will trigger various sporadic operations (e.g. upgrade, reconfiguration, redeplooy or backups) that will impact the execution of a running application. Continuous deployment practices make such sporadic operations much more frequent. Deployments of multiple times a day are becoming more common [1]. These sporadic operations trigger many alarms, mainly false positives. A normal practice is for operators to turn off monitoring during sporadic operations such as redeployment, reconfiguration and upgrade as a means of reducing false positives during the operation. However, this period is often the most vulnerable and such practices may mean missing important alarms. Since sporadic operations are happening much more frequently, manual re-adjustment is no longer possible. When change is the norm, anomaly detection relying on traditional norm baseline settings will not work.

Another challenge is the specification of monitoring parameters. Continuous change in the system infrastructure and the system itself complicates the setting of monitoring parameters. On the infrastructure side, even if you are requesting exactly the same virtual machine type, there is significant performance variance due to factors beyond your control, such as the CPU type you get [2]. As a consequence it makes sense to automate the configuration of alarms, alerts, and thresholds as much as possible. The setting of monitoring configurations is another operation process that can be and...
should be automated. This includes elasticity-aware automatic
deregistering/registering servers from/into monitoring system
and also automatic readjustment of monitoring parameters
based on operation context.

**Contributions:** The key contribution of our approach is the
use of process context to reduce the false positive alarms
during sporadic operations on cloud applications, which is
called Process-Oriented Dependability (POD)-Monitor. Process
context of an operation is the information such as the operation
process id, instance id, the start and end time of operations. Our
approach is non-intrusive and does not require any
modification to the existing system or the monitoring software
involved. Our contributions include 1) The direct use of
process context information to further examine threshold-based
alarms to reduce false positives; 2) The use of process context
as new features in machine learning, together with cloud
metrics, to better predict anomalies; 3) Features that allow
operators to use process context to automatically adapt the
alarm settings at runtime.

We conducted a series of experiments to compare the
effectiveness of using process context information to suppress
false alarms during the experiments with simultaneous
legitimate operations and faults being injected. The analysis
result shows that, comparing with threshold-based alarms, 1)
using process context directly improves the precision up to
0.226 (36.1% improvement); 2) using process-context trained
machine learning models improves the precision up to 0.421
(84.7% improvement).

The paper proceeds by introducing background in Section
II, followed the discussion of overall approach in Section III.
Section IV presents the realization of POD-Monitor. We
evaluate our approach in Section V, and discuss the related
work in Section VI and Section VII concludes the paper.

**II. BACKGROUND**

**A. Monitoring**

We consider two sources of monitoring data:

1) **Metric and Alarms:** Metrics are time series data that
show status of the monitored resources. Alarms are
configurations describing a significant state or state change
that are worth special attention. Typically, an alarm is set by a
threshold representing the boundary between “normal” and
“abnormal”. Alarms are triggered and alerts are sent when a
certain metric falls outside its “normal” range defined by the
threshold.

2) **Logs:** A log is a time series of events. Logs written by
the developers are usually for developer’s development-time
diagnosis rather than with operators and operation time in
mind. One of the motivations of the DevOps movement has
been to treat operators as first class stakeholders and this
means writing logs that they can use. The sources of these logs
include applications but also web servers, database systems
and operation systems. (Sporadic) operations themselves also
produce logs.

Currently, logs are usually used for error detection and
diagnosis. In our earlier work [3], we extracted process models
from operation logs and used the process models to help detect
errors early and perform automated error diagnosis. In this
paper, we report on using the process context to reinterpret
metrics data to suppress false alarms both directly and
indirectly through training machine learning models.

**B. Sporadic Operations and Monitoring**

Systems in the cloud are subject to sporadic changes due to
operational activities, which could be happening at the same
time. There is a sporadic nature to these operations as some are
triggered by ad hoc bug fixing and feature delivery while others
are triggered by periodic maintenance activities. As pointed out
by Jeff Dean from Google, even these periodic maintenance
activities may not be able to be synchronized [4] and
predictable. Some operations have significant impact on
monitoring metrics at both the instance level and the systems
level. For example, upgrading could affect overall system
availability depending on the upgrade strategy since an
individual instance may come and go or be rebooted causing
metrics anomaly. All these could be mis-interpreted as alarms
if the context of the sporadic operation is not considered. In the
mean time, true anomalies during the operation may be hard to
discern due to the many legitimate impacts on collected
metrics. Some other operations may have only minor impacts
on individual metrics such as CPU utilization being affected by
backup but may have bigger consequences in a larger scale
synchronous fashion [3].

A smart monitoring system should be able to use the
process context information of the operations to dynamically
interpret the metrics data to better set and raise alarms. This is
what motivates our approach.

One example of sporadic operation on Cloud application is
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 [1] without system downtime. In a rolling
upgrade [5], a small number of \( k \) instances at a time currently
running the old version are taken out of service and replaced
with \( k \) instances running the new version. Rolling upgrade is a
widely used method for upgrading instances [5]. Current
implementations of rolling upgrade vary. We base our
discussion on the method Asgard\(^1\) currently uses. Asgard is
Netflix’s customized management console on top of the
Amazon Web Services (AWS) infrastructure. Asgard does
rolling upgrade by terminating instances from the ASG, and
utilizing the ASG to launch new instances with the new
version. During a rolling upgrade, monitoring and anomaly
detection can be very challenging. First, a number of instances
are being taken out of service legitimately causing naïve alarm
settings to raise false alarms. Second, the impact on various
metrics could be big or small depending on \( k \) and the version
activation strategy. This makes specific alarm setting for rolling
upgrade highly dependent on the particular activities being
performed (the process context). Third, a true fault of an
instance during this period exhibits similar behavior and has

\(^1\) Asgard—https://github.com/Netflix/asgard
similar impacts as a legitimately killed instance. Finally, other simultaneous operations and varying traffic may confound any anomaly detection mechanisms.

C. Process Context

Our previous work [3] automatically mines process models from the logs produced by the operation. The mined process model is used by the log processor to match the log lines and extract the process context information from the log lines at runtime. Thus, we can know the exact start, progression and stop time of a sporadic operation or a step within the sporadic operation.

III. POD-Monitor Approach

Fig. 1 shows the workflow of POD-Monitor approach. We assume alarms are generated normally by a monitoring system even during sporadic operations. The first stage in developing the suppression logic is to train predictive machine learning models by using both cloud metrics (e.g., CPU, networks etc) and process context information from past execution traces. We then integrate the prediction model into the alarm suppressor component used at runtime. Our alarm suppressor has two ways to suppress false positive alarms, including using the process context directly and using the machine-learning model which is trained by the process context. Our approach in using machine learning is not to replace the existing monitoring and alarm setting facilities. Instead, the operators can choose to pass alarms raised by their monitoring system rather than requiring it to be used immediately.

D. Data Collection

We collected training data by running various operations on an existing system. The operations include rolling upgrade and other overlapping legitimate operations. The detailed experiment deployment and setting are discussed in Section V. We collected metrics data from CloudWatch as well as process context information from the log produced by the operations. CloudWatch provides the metric data in JSON file, which can be accessed through an API. The process context information is collected automatically from the log lines produced by the operations during experiments through distributed log processor agents deployed on the instances. For more details, please refer to our previous work in [3].

C. Machine Learning Strategy Selection

Anomaly detection is a classical machine learning classification problem [6]. We have chosen Support Vector Machine (SVM) [7] after some initial small-scale exploration. For example, we tried k-means for unsupervised classification. The best prediction results from k-means have precision of 0.704, and recall of 0.985. These results are bad as compared with the prediction result of SVM, the detail of which is shown in TABLE I. Besides, SVM is often recognized as one of the most efficient and accurate classification technologies. There are several existing tools that can be employed to do such training, and we use LibSVM toolbox in Matlab.

A. Machine Learning Challenges

We use AWS CloudWatch metrics, which provides more than 40 types of metrics. The CloudWatch metrics represent time-ordered sets of data points. CloudWatch does not provide real-time information, as under typical settings the granularity of data is per-minute. There could be a significant time delay between when a fault occurs and when an abnormal value is reflected in CloudWatch metrics and subsequent alarms if any. Conversely, an alarm or abnormal values could last for a certain time period even after the underlying faults have been recovered.

Fig. 2 gives a set of sample metric data of a rolling upgrade with two legitimate operations: CPU-intensive operation (e.g., log compression) and Network-intensive operation (e.g., data reconstruction). The start time and end time of CPU-intensive operation are indicated by dots with a value of 100 at the secondary vertical axis, the start time and end time of Network-intensive operation are indicated by dots with a value of 90 at the secondary vertical axis, and the fault injection time is indicated by dots with a value of 110 at the secondary vertical axis. It shows that the impact of fault injection on the metrics cannot be easily detected by human eyes or simple thresholds due to the interference of simultaneous legitimate operations on real faults. Moreover, cloud systems have inherent fault tolerance features that confound anomaly detection further.

B. Data Collection

We collected training data by running various operations on an existing system. The operations include rolling upgrade and other overlapping legitimate operations. The detailed experiment deployment and setting are discussed in Section V. We collected metrics data from CloudWatch as well as process context information from the log produced by the operations. CloudWatch provides the metric data in JSON file, which can be accessed through an API. The process context information is collected automatically from the log lines produced by the operations during experiments through distributed log processor agents deployed on the instances. For more details, please refer to our previous work in [3].

C. Machine Learning Strategy Selection

Anomaly detection is a classical machine learning classification problem [6]. We have chosen Support Vector Machine (SVM) [7] after some initial small-scale exploration. For example, we tried k-means for unsupervised classification. The best prediction results from k-means have precision of 0.704, and recall of 0.985. These results are bad as compared with the prediction result of SVM, the detail of which is shown in TABLE I. Besides, SVM is often recognized as one of the most efficient and accurate classification technologies. There are several existing tools that can be employed to do such training, and we use LibSVM toolbox in Matlab.

---

TABLE I. SVM Results

<table>
<thead>
<tr>
<th>Metric Type</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>0.704</td>
<td>0.985</td>
</tr>
</tbody>
</table>

---

svmlib – http://www.csie.ntu.edu.tw/~cjlin/libsvm/
D. Feature Selection and Engineering

CloudWatch provides basic statistics of each of its metrics, for example, average, minimal, and maximum. We consider each of the listed statistic of a specific metric as a separate feature. Thus, there are 18*3 features from CloudWatch metrics. The ranges of all the features are rescaled to [0, 1] through normalization. In addition, we also have 7 additional features of process context to represent different operations and activities within operations. Each data instance represents the metrics information for a single time point.

E. Training

We have collected 2136 data instances from CloudWatch, and we divide the data into one training set and one predicting set. We determine the labels based on whether there is fault occurrence in each data instances. However, whenever a fault occurs, the impact on those metrics appears later than the occurrence and lasts for a while. Here we introduce the concept of “time window”, the time period from the current time point of each data instance. For example, when the time window size is 3, if there is a fault recorded by the current data instance, the instance is an alarm and the next two data instances are also alarms. The window size should be decided in terms of the system attributes, such as the type of instances, the size of virtual machines and images, and the configurations and so on. For example, if a fault occurs in a micro-instance and results in a replacement of this instance, the time between the loss and the recovery of the instance follows a normal distribution with the mean around 3 minutes. Thus, 3 minutes can be a reasonable setting of window size. We use this method to label all the data instances. In the training procedure, the kernel function we selected is RFB (Gaussian kernel). We identified the optimal values of \( g \) (Gamma) and \( c \) are 0.1 and 32 through 10-fold cross-validation.

F. Prediction

After an SVM-model is trained, the trained model can be used for online prediction. It is important to note that due to the window mechanism discussed earlier, when the model predicts an alarm in one particular minute, the real answers can be in the minutes next to the predicted minutes. Thus we have developed an eventual alarming policy considering this uncertainty together with the SVM-model prediction. The size of the window is adjustable and has an impact on the prediction results, which are shown in TABLE I. We have chosen a window size of three in our approach. The results in TABLE I also show that our predictive model with process context as new features could achieve a reasonable precision and recall rate.

<table>
<thead>
<tr>
<th>Effective Interval</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.929</td>
<td>0.863</td>
</tr>
<tr>
<td>2</td>
<td>0.760</td>
<td>0.838</td>
</tr>
<tr>
<td>1</td>
<td>0.282</td>
<td>0.657</td>
</tr>
</tbody>
</table>

Importantly, removing the process context features will decrease the precision and recall rate. However, we observe that the decreases are related to the frequency of the faults being injected. When the faults are injected at about 1 every 3 minutes, the decrease of precision is about 0.12. When the faults are injected at 1 in every 15 minutes, the decrease of precision is about only 0.02. This is largely due to the manifestation of the faults and the legitimate killing of instances are almost identical and can not be discerned if they are too close to each other.

IV. IMPLEMENTATION OF POD-MONITOR

POD-Monitor provides three ways to suppress false alarms: using process context directly to suppress alarms, using machine-learning model to suppress alarms, or adapting alarm threshold according to the process context. At runtime while sporadic operation are executing, operators have options. If operators have threshold-based alarms, they do not have to change anything or disable their alarms. The raised alarms will be passed into our suppression system and we will use process context directly and the machine learning models to re-interpret these alarms to reduce false alarms. If operators would like to use the process context to dynamically adjust alarm settings to remove initial false alarms directly, they can do so as well through the alarm setter component.

Fig. 3 gives an overview of the POD-Monitor architecture. The filled boxes represent the components of POD-Monitor, while the unfilled boxes represent third-party components. A cogwheel is a log processor. The grey arrows show the flow of alarms, and the black arrows show the flow of other information. At runtime, alarms caused by cloud resources are
one primary source of information. Alarms could be generated by monitoring mechanisms provided by cloud infrastructure, such as AWS CloudWatch or generated by general monitoring framework used by cloud applications, such as Sensu\textsuperscript{3} or Nagios\textsuperscript{4}. After Alarm Collector collects an alarm generated by the monitoring system, the alarm goes to Alarm Suppressor, which decides whether the alarm should be suppressed. Alarm Annotator annotates the alarm with the current process context, the suggestion given by the Alarm Suppressor, and forwards the alarm to operators through email or other means.

Logs from operations are the other primary source of information. Some logs are generated by the cloud infrastructure while other logs are generated by operations. As we operate in the cloud, some logs are hidden by cloud providers but can be requested through API calls. The logs are collected and filtered by log processors. The process context repository is updated by the process context information extracted from the log lines.

A. Log processor

We use Logstash to implement our log processors. Logstash\textsuperscript{5} is an open source log management tool implemented in Java. We deploy a decentralized Logstash agent on each of the instances that we are monitoring, and with every operations running on the side. When Logstash agent receives a log line, the log line goes through a sequence of plugins defined in the agent as a pipeline. A Logstash agent uses a configuration file to specify the the source of the log files, the filters used to process the log lines, and the output of the log lines.

Alarm collector and Alarm Setter are implemented as RESTful Web Service using RESTlet\textsuperscript{6}, a RESTful Web API framework for Java. Alarm Collector and Alarm Setting are the two components that are specific to the monitoring system.

We selected AWS CloudWatch as the first monitoring system to implement our idea. As shown in Fig. 4, CloudWatch can monitor AWS cloud resources, such as EC2 and Amazon RDS DB instances, and the application running on AWS infrastructure. In our implementation, the action is a notification sent to an Amazon Simple Notification Service (SNS) topic. Our Alarm Collector subscribes to the SNS topic and receives CloudWatch alarms as HTTP request.

![Fig. 3. POD-Monitor Architecture Overview](image)

![Fig. 4. CloudWatch Alarm Manipulation](image)

We don’t modify the existing alarms setting. Every time an operation starts, or an action within operation starts, we set operation-specific and action-specific alarms through CloudWatch API according to the knowledge-based operation/action features. Furthermore, we delete the operation-specific and action-specific alarms accordingly when the operation/action completes. Currently, the operation/action features are manually defined based on our experience and understanding of operations. We can then annotate the alarm with additional information or suppress it.

V. EXPERIMENTAL EVALUATION

A. Experiment Deployment

We chose AWS as the cloud infrastructure. The experiment is deployed on an Auto Scaling Group (ASG) containing 8 identical EC2 instances that can be upgraded in a rolling upgrade fashion. We used Siege\textsuperscript{7}, a http load testing and benchmarking utility to create workload on top of a background workload for our experiments. We performed rolling upgrade on the EC2 instances through heavily-baked upgrading approach, which uses pre-baked images to replace existing VM instances [8]. We used Asgard to do rolling upgrade.

To simulate a complex ecosystem, we ran another two legitimate background maintenance operations in parallel to rolling upgrade – a CPU-intensive operation (representing operations like log compression) and a Network-intensive operation (representing operations like remote backup or data reconstruction). Both the CPU-intensive and Network-intensive operations produce logs as well. In addition, a fault injection program is running on the side to inject EC2 instance fault through randomly killing instances within the ASG during experiments.

\textsuperscript{3} Sensu – http://sensuapp.org

\textsuperscript{4} Nagios – http://www.nagios.org/

\textsuperscript{5} Logstash – http://logstash.net

\textsuperscript{6} RESTlet – http://restlet.org

\textsuperscript{7} Siege – http://www.joedog.org/siege-home/
B. Experiment Methodology

We conducted experiments to run rolling upgrade with different legitimate operations running on the side. The setting of our experiments is aimed to cover the possible combinations of different operations and fault injections as many as possible and also to face worse cases. We adjusted the starting time of the operations to intertwine the operations in different ways. Both the CPU-intensive operation and Network-intensive operation are injected every 2 minutes, and lasts 2 minutes. Instance faults are injected randomly. The granularity of the rolling upgrade is 2 instances, which means there are at most 2 instances being upgraded at anytime. Eventually, we did 72 experiments, which lasts 2180 minutes in total. Note that the experiment setting twists rolling upgrade, other operations, and fault injection to be more complicated than most possibilities in reality. If we can achieve sounding results on this, we can deal with the most real cases easily.

We used the data collected during the 72 experiments to do offline data analysis and compare the results with those from the traditional threshold approach, threshold approach with direct process context-based alarm suppression and threshold with machine learning based alarm suppression.

C. Experiment Results

We used alarm precision, alarm recall, and alarm F-score as the three metrics to measure the threshold-based monitoring approach. F-score is the harmonic mean of precision and recall. We use the three metrics to compare different results.

1) Threshold-only approach: We show the result of CPU Utilization metric. All the other metrics have similar results. In CloudWatch, an alarm is triggered only once when the metric crosses the threshold. When we do the data analysis, we take into account the all the data that is larger or smaller than the defined threshold as alarms depend on the condition of the threshold.

Fig. 5 shows the precision of alarm, recall of alarm and F-score of alarm on different CPU thresholds. When the CPU threshold grows from 10% to 95%, the precision of alarm increases from 0.485 to 0.852, while the recall of the alarm decreases from 0.996 to 0.043. This is because the lower the CPU threshold, the more false alarms it may cause. In the similar way, the higher the CPU threshold, the more true alarms it may miss out. The F-score of alarm decreases from 0.653 to 0.0824 as the CPU threshold increases. When the CPU threshold is 10%, the F-score is maximal. As mentioned earlier, in reality, very low or very high CPU thresholds do not make sense in practice.

<table>
<thead>
<tr>
<th>CPU Usage</th>
<th>Threshold approach using CPU only</th>
<th>Threshold approach with process-context suppressing using CPU only</th>
</tr>
</thead>
<tbody>
<tr>
<td>60%</td>
<td>0.541</td>
<td>0.608</td>
</tr>
<tr>
<td>65%</td>
<td>0.559</td>
<td>0.626</td>
</tr>
<tr>
<td>70%</td>
<td>0.577</td>
<td>0.673</td>
</tr>
<tr>
<td>75%</td>
<td>0.608</td>
<td>0.710</td>
</tr>
<tr>
<td>80%</td>
<td>0.626</td>
<td>0.852</td>
</tr>
</tbody>
</table>

2) Using process-context for alarm suppression: The main objective of our approach is to reduce false positive alarms by using process context. Thus, we compare the precision of the traditional threshold approach, and the threshold approach with direct process-context based alarm suppression in this section. The comparison is shown in TABLE II. The second column gives the precision of CPU utilization using threshold-approach. The third column gives the precision of CPU utilization using the threshold-approach with process context alarm suppression. The result shows that using process context alarm suppression improves the precision in a certain extent. However, the overall result of direct process context alarm suppression (F-score) is not always better than the threshold-approach. The maximum improvement of precision is 0.226 at threshold 80%, the minimal improvement is 0.067 at threshold 60% and 65%, and the average improvement is 0.11. Note that we remove the values at very low and very high thresholds as they are not realistic.

3) Process-context trained machine learning model for alarm suppression: Fig. 6 gives the comparison between threshold-approach with and without machine learning based alarm suppression on the metric of CPU utilization. It shows that using machine-learning model with process context to double evaluate the alarms could increase the precision of CPU Utilization at all the different thresholds setting from

Fig. 5. Precision and Recall of CPU Utilization-based Threshold

Fig. 6. Comparison between Threshold-approach with and without Machine-learning based alarm suppression on CPU Utilization Metric
0.171 (at 90%) to 0.413 (at 35%). The average improvement of precision is 0.353. The recall of the alarm increases as well.

D. Discussion

There are some limitations to our approach. First, we evaluated our approach under three sporadic operations, which may not be able to represent the diverse real world operations. On the other hand, we introduced significant confounding factors such as large workload variation, high fault frequency and operation frequency. They may make the problem more difficult. Second, the effectiveness of the approach is also related to fault frequency and monitoring intervals. We didn’t explore these two factors systematically. Third, using process context directly and using machine learning models (trained using process context) are both effective. It is not clear whether using machine learning adds significant values considering the cost involved. We will explore further about the strength and weakness of the two approaches and the applicable scenarios.

VI. RELATED WORK

In existing work of threshold-based methods [9], thresholds are set on system and application performance metrics based on historical observations of an application behavior and an alarm is raised if any threshold is violated. This approach forms the basis of many commercial (like IBM Tivoli, HP Openview) and open source (like Ganglia, Nagios) monitoring tools [9]. Our naïve threshold based method is more sophisticated and can cater for process context.

Existing works [10-15] in machine learning for anomaly detection use only cloud metrics and are used during normal operation (rather than sporadic operation). Those works normally use statistical techniques to build a performance model of the system under normal behavior and flag deviations as anomalies. Application-based correlations as well as peer-based correlation methods have been proposed as effective ways to capture the performance invariants, and variations from the modeled correlation are being treated as anomalies. Correlation invariants are effective, but are expensive to learn, and require large training data, especially for non-linear correlations. Our use of machine learning without using process context essentially follows this type of approaches. However, it shows using process contexts as features can improve the overall results.

Industry is increasingly using “adaptive monitoring” to dynamically adjust monitoring threshold or completely move away from thresholds [16]. However, it is largely based on incorporating more ad hoc knowledge flexible monitoring code or trying more sophisticated anomaly detection algorithms. Incorporating more domain knowledge is time consuming and fragile to future changes. The use of more sophisticated algorithms is good but these algorithms essentially do not consider the process context information.

VII. CONCLUSION

Detecting true anomaly in a constantly changing environment is always difficult. Monitoring is usually designed for normal operation time using historical base line. They are often turned off when the system is undergoing significant changes through sporadic operations such as upgrade, redeployment and backup. We allow alarms from monitoring system to be enabled during sporadic operation and then use the process context to further suppress false alarms. Our results show that the process context significantly helps in reducing false positives.

ACKNOWLEDGMENTS

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