Semantic Aware Online Detection of Resource Anomalies on the Cloud

Arnamoy Bhattacharyya *, Seyed Ali Jokar Jandaghi*, Stelios Sotiriadis* and Cristiana Amza*
*Department of Electrical and Computer Engineering
University of Toronto, Toronto,Canada

Abstract—As cloud based platforms become more popular, it becomes an essential task for the cloud administrator to efficiently manage the costly hardware resources in the cloud environment. Prompt action should be taken whenever hardware resources are faulty, or configured and utilized in a way that causes application performance degradation, hence poor quality of service. In this paper, we propose a semantic aware technique based on neural network learning and pattern recognition in order to provide automated, real-time support for resource anomaly detection. We incorporate application semantics to narrow down the scope of the learning and detection phase, thus enabling our machine learning technique to work at a very low overhead when executed online. As our method runs “life-long” on monitored resource usage on the cloud, in case of wrong prediction, we can leverage administrator feedback to improve prediction on future runs. This feedback directed scheme with the attached context helps us to achieve an anomaly detection accuracy of as high as 98.3% in our experimental evaluation, and can be easily used in conjunction with other anomaly detection techniques for the cloud.

I. INTRODUCTION

Stateful online distributed systems have become central to many modern business operations. More and more people trust cloud services offered by well-known providers, such as, Amazon, or large social network sites, such as Facebook, with running critical enterprise and data processing applications and/or with storage of personal profiles and other data they care about.

Behind the scenes, the dependability and reliability needs of these systems pose unprecedented problems. The support infrastructure for such sites, e.g., for large social network sites, such as, Facebook’s HBase/HDFS solution [?], as well as for large-scale data processing, such as, Cassandra [?] rely on traditional, relatively simple, performance debugging and reliability solutions for stateful services e.g., logging and 3-way replication. These traditional reliability solutions assumed that stateful components were few e.g., a monolithic database system, for which failures were rare, and that these few components and their failures occurred in somewhat predictable component states, which were monitored directly by humans.

These assumptions do not currently hold in stateful, large-scale distributed systems, such as, HBase, HDFS, or Cassandra, although most customer expectations have not changed. In these large-scale systems, stateful components are expected to be many, and failures are expected to be the rule rather than the exception; for example, one hardware failure per data center, per day is commonly reported [?]. Moreover, the necessary maintenance activities for monitoring, diagnosis, inspection or repair can no longer be handled through frequent human intervention. Vast amounts of logs and monitoring data are generated, e.g., 1 Terrabyte of logging data per day is typical of a small to medium-size 100 node Cassandra cluster. Therefore, a system administrator cannot catch up with inspecting the vast amounts of data produced by all stateful components. Even if monitoring data could be read by operators, humans may no longer be able to predict or recognize all possible combinations of states and situations leading to unrecoverable errors for the many components of large-scale sites.

Newer solutions that take into account the fact that Clouds contain a large number of components provide automated log and monitoring data collection and some anomaly diagnosis services [?], [?], [?], [?], [?], [?], [?]. However, in order to be able to make sense of these errors, either the user’s expectations need to be pre-specified, or significant automated post-processing overhead needs to be involved.

More recently, several automatic, purely black-box anomaly detection solutions have been developed for automating pattern recognition in monitoring data e.g., through deriving statistical pair-wise correlations, or by peer comparison [?], [?], [?], [?]. Other black-box solutions reverse match the anomalous patterns to known code patterns in order to help the users in their anomaly diagnosis [?], [?], [?]. These black-box solutions are more flexible and applicable to large-scale systems, but, they still incur significant storage and processing overhead. As before, anomaly diagnosis is typically performed off-line, post-facto, at which point vast amounts of information has already been collected.

In this paper, we leverage recurrent neural networks (RNN) [?] for learning and recognizing resource consumption patterns online, in real-time. Our method uses minimal instrumentation in order to capture and label the application contexts during which the resource monitoring occurs. For the purposes of this paper, these contexts consist in the known workload mix type and phase that was used during the monitored application runs. Alternatively, we could use as application context the application method, module, or other clearly identifiable piece of code, or semantic scope of the application, and its input values, or the application code running the same method/module, independent of parameter values.

Specifically, we capture and investigate the throughput and
latency metrics at the application level, as well as the CPU, memory and disk consumption monitoring data, and tag them with the application context in which they occur at their collection point. Then, as data is collected, our RNN machine learning technique is deployed in order to learn mathematical models, per context. Finally, at the point of root-cause analyses by human inspection, whether these are performed on the live system, or off-line, only the contexts that statistically diverge from the learned model are signalled to the user and, for each, we provide context-rich problem digests.

Our technique assumes that the workload types of applications are known and/or that open-source systems are used in the Cloud software stack. Specifically, we monitor Apache Cassandra\(^1\) and MongoDB\(^2\) with regards to their resource consumption and throughput for known workload mixes involving read and write operations.

Instead of searching through all possible patterns, in order to learn and distinguish baselines versus anomalies, in vast amounts of unlabeled data, our RNN-based machine learning technique benefits by our grouping the monitoring information by context. By collecting information pre-sorted by context(s), we thus enable both i) machine learning and ii) user to perform the tasks that they perform the best as follows. The machine can process large amounts of data to generate the typical wavelet/signal and/or statistical attributes i.e., distribution, standard deviation, etc, for the various resource consumption patterns of each application context, or for sets of related application contexts. Based on the extracted normal signals by context, the machine can also generate compact synopses for any suspected anomalies or corner cases only, at run-time, while remembering only the learned model for the normal behavior of each context. In his turn, the user can provide feedback for selective supervised learning - normal versus anomalous - based on the few contexts which the machine has automatically detected as unusual, or for specific known error cases which the machine may have incorrectly considered normal.

After refinement of predictions, based on user feedback, our method is expected to have two advantages: i) it will reduce the high storage or performance overheads in the case where a mathematical model can be derived for each per-context application pattern or signal and all information that matches the “normal” pattern of a context can be discarded instead of written to disk and ii) the user will have pre-sorted, pre-classified, condensed and semantically tagged data to investigate in order to perform root cause analysis.

In our experimental evaluation, the applications of interest are deployed in cloud virtual machines (VMs) using OpenStack\(^3\). We utilize an intensive workload generator, the Yahoo cloud serving benchmark \([4]\), to stress the system with a large variety of workloads similar to real world scenarios.

We use two scenarios, of fault injection and resource interference, respectively, in order to explore the behavior of two well known cloud software packages, Apache Cassandra and MongoDB, under both normal and anomalous environment and workload conditions. In the fault injection experiment, we inject disk I/O delays, emulating a faulty or congested disk behavior, for a certain period of time, during separate experiments, with Cassandra or MongoDB. In the resource interference experiment, we run the regular Cassandra code in one VM together with a second VM running a misconfigured Cassandra VM acting as a disk hog, which produces disk interference for the regular Cassandra VM. Both the disk I/O delay and the disk hog perturbations produce changes in overall VM resource usage, as well as significant application throughput drops for the regular Cassandra and MongoDB workloads. In our experimental evaluation, we show that our method allows us to detect normal versus anomalous resource consumption patterns with 98.3% accuracy.

By offering context based anomaly extraction with minimal processing cost, we anticipate that our solution will promote augmenting routine system monitoring with real-time statistical analysis for context-aware anomaly detection. Moreover, our technique can be easily used in conjunction with existing, orthogonal anomaly detection methods, such as previous methods for processing application text messages from logs \([2, 3, 4, 5]\).

II. MOTIVATIONAL EXPERIMENT

Figure \(??\) demonstrates a representative scenario for anomaly detection and/or diagnosis. We monitor the average total throughput of a four node Cassandra cluster running the Yahoo! Cloud Serving Benchmark (YCSB) workload\(^4\) over time.

The Cassandra cluster registers a relatively stable high throughput of about 6000 operations/second initially, until around time 70 (seconds) when we start perturbing one node of the system. YCSB has an insert and update phase. Towards the end of the YCSB update phase, for a period of 100 seconds, we inject disk delays on read and write operations as follows: 50% of Cassandra reads and 50% Cassandra writes to/from disk on the affected node are delayed by 50 seconds each.

Although we inject the delays into a fraction of the operations of only one of the cluster nodes, we observe that (i) the throughput of the whole cluster drops significantly, as shown in Figure \(??\), (ii) during the drop, there is an even load distribution among the nodes and (iii) Cassandra can not recover its throughput from the drop throughout our disk delay injection period.

This significant and sustained throughput drop is a typical scenario of performance anomaly where, even if one of the causes is known or suspected, a thorough investigation of all related monitoring and log data showing anomalies would help.

Since Cassandra is a complex, resource intensive distributed system, we introduce an anomaly detection technique, which

---

2. https://www.mongodb.com
3. https://www.openstack.org
helps with the above investigation in two ways: i) by automatically identifying normal versus anomalous patterns in the resource consumption monitoring data associated with the application run, on-the-fly, and ii) by identifying the application contexts, such as, the workload characteristics and/or workload phase during which the anomalies occurred.

![Huge performance degradation of a Cassandra Cluster for a faulty disk.](image)

Figure 1: Huge performance degradation of a Cassandra Cluster for a faulty disk.

In the following, we introduce our context-aware statistical learning method.

III. INFERRING ANOMALIES FROM RESOURCE USAGE MONITORING PER CONTEXT

Our key idea is to add application context to the machine learning system that is in charge of differentiating normal versus anomalous resource consumption patterns. Under the assumption that repeatable workload patterns will give rise to similar resource consumption patterns, training on data already classified by context reduces the training time by helping the classifier to discover the similar patterns in the data. We use recurrent neural networks (RNN) [8] for this purpose, because they are widely used in pattern recognition.

Figure 2 shows the complete workflow of our machine learning system. During training, the deep learning RNN engine is presented with monitoring data pre-labeled with the representative workload type and phase under which it was gathered. Over time, the engine learns the statistically expected resource consumption patterns, per resource, and per workload. All pre-labeled learned patterns are stored in a wavelet database. After a period of training, new data labelled with its workload patterns is presented to our monitoring and statistical learning system. At this point, we match the workload patterns with those stored in the wavelet database and we determine if the new resource consumption patterns match the corresponding resource consumption wavelets from the database. We use a technique called Dynamic Time Warping (DTW) [7] for this purpose. If DTW does not find a match, then an anomaly is presented to the user together with the workload type and phase in which it was detected. We can incorporate user feedback in case the unusual pattern is in fact not an anomaly. The user can also input resource consumption patterns associated with known errors to the deep learning system, which can potentially correct misclassifications by the system of known faulty patterns. In the following, we present our semantic aware resource monitoring, deep learning with RNN, and wavelet matching using DTW in more detail.

A. SEMANTIC AWARE RESOURCE MONITORING

Our resource monitoring module samples resource consumption periodically, at a given sampling rate. Over time, it gathers training data, which is periodically fed to our deep learning system. Our data collection methodology considers the semantic of the particular workload being run on the cloud. Semantically specifying the scope within which different resource consumption data is collected greatly helps the machine learning algorithm to recognize patterns and detect anomalies with a much lower overhead.

The resource monitoring module shown in pseudo-code in Algorithm 1 receives and initializes the workload configuration, including the YCSB dataset parameters: workloadName, recordCount, OperationCount, ReadPortion, UpdatePortion, ScanPortion, InsertPortion, and other variables related to the number of threads to be executed, the resource usage and the sampling rate.

After the initial state is captured, the workload characteristics and phase names are used as context parameters in the execution of various application phases e.g., the YCSB data load (insert), the data read and update phases of the workload. At the end, the monitoring module returns the resource usage results correlated with the workload characteristics and with each workload execution phase (e.g., insert, read and update) thus providing pre-labeled data for pattern recognition.

Algorithm 1 Context Capture during Resource Monitoring

**Requires**
- workloadConf: workloadName, RecCount, OpCount, ReadPortion, UpdatePortion, ScanPortion, InsertPortion
- NumThreads: Number of threads to be executed
- SamplingRate: The sampling rate
- ResourceRetrievalFeatures: CPU, Memory, Disk usage, etc.

1. **procedure** RUNYCSB(workloadConf, NumThreads)
2. **startCassandra()**
3. **startWorkload(ResourceRetrievalFeatures, SamplingRate, NumThreads)**
4. **load(workloadConf,“ld-phase”, RecCount)**
5. **read(workloadConf,“rd-phase”, OpCount)**
6. **update(workloadConf,“up-phase”, OpCount)**

**Returns** ResourceUsageResultsPerPhase
B. Deep Learning from the monitored data

Our next step is to identify a pattern from the resource monitoring data for a given semantic scope (in our case, the workload characteristics and phase). After the semantic aware resource monitoring in the first step, we have a collection of \( n \) training wavelets from \( n \) runs of each workload \( w_i \). The output from this stage will be a pattern \( P_i \), one for each monitored resource for a given workload. The duration \( d \) for a run of a given workload is fixed across all the training runs. The machine learning produces a representative resource usage wavelet, with a duration \( d \). This wavelet aggregates all the minor fluctuations the \( n \) training runs have for different resource usages.

To generate this representative wavelet, we use deep learning methods, specifically Recurrent Neural Networks [\( ? \)].

We have a sequence of inputs (the wavelets from \( n \) training runs) for an output \( y^j \) for a given resource, so we denote the inputs by \( x^j \) for the \( j \)th input (\( j = 1, 2, \ldots, n \)). The corresponding \( j \)th output is calculated via the following equation:

\[
y^j = W x^j + W_r y^{j-1}
\]

We have a weight matrix \( W_r \) which incorporates the output at the previous step linearly into the current output.

In most common architectures, there is a hidden layer which is recurrently connected to itself. Let \( h_j \) denote the hidden layer at timestep \( j \). The formulas are then:

\[
h^0 = 0
\]

\[
h^j = \sigma(W_1 x^j + W_r h^{j-1})
\]

\[
y^j = W_2 h^j
\]

Where \( \sigma \) is a suitable non-linearity/transfer function like the sigmoid. \( W_1 \) and \( W_2 \) are the connecting weights between the input and the hidden and the hidden and the output layer. \( W_r \) represents the recurrent weights.

The RNN workflow is demonstrated in Figure ??

Running the RNN over the \( n \) training wavelets generates a prediction model. Using this model, we perform a prediction for the input \( x_n \) which is the last captured wavelet in the training data. The forward prediction of length \( d \) for the last training wavelet generates the representative wavelet \( P_i \) for that workload type and phase. In the next step, this wavelet is matched against future test wavelets to differentiate between a normal and an anomalous execution.

C. Wavelet Matching Using Dynamic Time Warping

At the end of the machine learning algorithm, we form a database that consists of all the representative signals for all the tested workloads, for each monitored resource. This mapping information is necessary for testing new data for anomalies in
an online fashion. When new data arrives, we know the workplace type from the semantic analysis. Then we find the relation between the test wavelet \( T_i \) and the representative wavelet \( P_i \) for that given workload that is stored in the wavelet database. We use a widely recognized signal processing technique called dynamic time warping (DTW) \([2]\) to find the relation between the test wavelet and the representative wavelet.

Both the wavelets will have \( d \) data points in them as described in the previous section. Suppose the wavelets \( T_i \) and \( P_i \) consist of the following sampled values:

\[
T_i = t_1, t_2, \ldots, t_d
\]

\[P_i = p_1, p_2, \ldots, p_d\]

To align these two sequences using DTW, we first construct a \( d \)-by-\( d \) matrix where the \((i \text{th}, j \text{th})\) element of the matrix corresponds to the squared distance, \(d(t_i, p_j) = (t_i - \bar{p}_j)^2\), which is the alignment between points \( t_i \) and \( p_j \). To find the best match between these two sequences, we retrieve a path through the matrix that minimizes the total cumulative distance between them. In particular, the optimal path is the path that minimizes the warping cost:

\[
DTW(T, P) = \min \left\{ \sum_{k=1}^{K} w_k \right\}
\]

where \( w_k \) is the matrix element \((i, j)_k\) that also belongs to the \(k\)-th element of a warping path \( W \), a contiguous set of matrix elements that represent a mapping between \( T \) and \( P \).

This warping path can be found using dynamic programming to evaluate the following recurrence.

\[
\begin{align*}
\gamma(i, j) &= d(t_i, p_j) + \min \{ \gamma(i-1, j-1), \gamma(i-1, j) \}, \\
\gamma(i, j-1) &= \end{align*}
\]

where \( d(i, j) \) is the distance found in the current cell, and \( \gamma(i, j) \) is the cumulative distance of \( d(i, j) \) and the minimum cumulative distances from the three adjacent cells.

At the end of the DTW algorithm, we have a minimum distance \( dist_{\min} \) that will warp \( T \) to \( P \). The lower the distance, the more similar the signals are.

To use the DTW distance to measure differences between normal and anomalous signals, we use the following method.

During the training phase, we calculate the DTW distance between each of the training wavelets and the representative wavelet and calculate the maximum distance, we call it \( \text{thres}\_\text{normal} \). This measure tells us the maximum deviation of a normal flow from the representative workflow. During the testing phase, if the calculated DTW distance is larger than the \( \text{thres}\_\text{normal} \) for a test wavelet, as calculated above, we flag the execution as anomalous.

As we are monitoring multiple resources, anomaly in any one resource consumption patterns will generate alarms. In the alarm information, we capture all the resource utilization patterns that are anomalous. This feedback gives the user better understanding on what is wrong.

### D. Dealing with False Alarms

There can be scenarios where, due to unexpected system activity, a normal execution still differs from our representative wavelet by more than the maximum DTW distance. In that case, we can incorporate feedback from the user to enhance our machine learning model (step 6 in Figure ??). We buffer the wavelet for any workflow we have detected as anomalous. Once the user tells us that our prediction is wrong, we feed the wavelet to our RNN and modify the representative workload to integrate characteristics from the new wavelet, including adjusting the maximum allowed deviation of the normal pattern. This method allows us to refine our prediction in case of false positives (where a normal execution is wrongly flagged as anomalous).

Similarly, user feedback is used to correct false negatives when resource monitoring patterns for certain workloads are not flagged as anomalous by our system, even if they actually are. For known errors, or whenever a new anomalous pattern is detected by the user, we store the user-discovered patterns for anomalous executions of a given workload pattern in our wavelet database. Whenever new wavelets are presented to our RNN system thereafter, we compute the DTW distance from both the normal pattern and the user-directed anomalous patterns. We flag a new wavelet as anomalous if its maximum distance to any of the anomalous patterns is smaller than its distance to the normal pattern.

### IV. Experimental Evaluation

Our experimental platform includes a four node Cassandra cluster configuration that is deployed in an OpenStack system. We use Apache Cassandra 2.0.16 and MongoDB Amazon 3.2.6 for running our experiments. Also, we use the YCSB Cassandra workloads (workload-a and workload-b) \([2]\). Workload-a is an application example for a session store recording recent actions and workload-b is an application example of photo tagging; adding a tag is an update operation, but most operations are to read tags. We set the number of threads to 5, the record and operation counts to 500000 and we use default values for all other variables. The monitoring sampling rate for the resource usage is set to a second. We use the following two anomaly scenarios for our experiments:

1) **Disk Delay:** In this anomaly scenario, we emulate a faulty disk with the help of the systemtap tool \([?]\). During the execution of a workload in Cassandra or MongoDB, we inject a delay of 50 seconds each in 50% of the reads and 50% of the writes to disk. We keep injecting the delay for a period of 10 seconds. This scenario is similar to the more extensive disk fault injection scenario we presented in section ?? where we motivated the investigation of a surprisingly high throughput drop in Cassandra associated with temporary disk delays.
2) **Resource Interference:** We co-locate two VMs running Cassandra workloads on the same physical machine. One of the workloads is acting as a disk hog, which is gradually increasing its disk utilization, and eventually exceeding its predeclared requirements for disk usage and causing resource interference to the other workload. Detecting this type of anomaly may help guide VM consolidation decisions. We use 2 identical medium size VMs (4 Cores, 4GB Ram, 40GB HD) deployed in OpenStack. The physical machine on which both VMs are placed has a Xeon(R) CPU E5-2650 2.00GHz (2x 8 Cores), 32GB of RAM.

For each of the above two anomaly scenarios, we have 20 normal runs and 20 runs with anomaly inserted for both the database systems. Each training run contains a complete run of the respective YCSB workload. The neural network is trained using the sampled data points from the 20 training runs and it extracts a representative wavelet for each workload. In the next step of training, we build the wavelet database by computing the DTW distance between the representative workload and the training wavelets from the 20 runs of each workload and storing the maximum distance, the workload type and the representative wavelet. The training time is around 1 second per training run, on average, for both Cassandra and MongoDB, thanks to our context aware machine learning method.

**A. Cassandra Results**

![Figure 4: Training wavelets and a test run showing the predicted wavelet(dotted) overlayed with a new run (solid) of the same workload, not seen during training for a Cassandra workload.](image)

Figure 4: Training wavelets and a test run showing the predicted wavelet(dotted) overlayed with a new run (solid) of the same workload, not seen during training for a Cassandra workload.

![Figure 5: The DTW distance helps to detect an anomalous run (grey) from a normal run (light grey) for disk delay anomaly in Cassandra, injected 55-65 sec.](image)

Figure 5: The DTW distance helps to detect an anomalous run (grey) from a normal run (light grey) for disk delay anomaly in Cassandra, injected 55-65 sec.

![Figure 6: The DTW distance helps to detect an anomalous run (grey) from a normal run (light grey) for resource interference anomaly in Cassandra. Although the disk hog is active during the whole run, its disk utilization grows gradually and approaches maximum during the last third of the run.](image)

Figure 6: The DTW distance helps to detect an anomalous run (grey) from a normal run (light grey) for resource interference anomaly in Cassandra. Although the disk hog is active during the whole run, its disk utilization grows gradually and approaches maximum during the last third of the run.

For each of the above two anomaly scenarios, we have 20 normal runs and 20 runs with anomaly inserted for both the database systems. Each training run contains a complete run of the respective YCSB workload. The neural network is trained using the sampled data points from the 20 training runs and it extracts a representative wavelet for each workload. In the next step of training, we build the wavelet database by computing the DTW distance between the representative workload and the training wavelets from the 20 runs of each workload and storing the maximum distance, the workload type and the representative wavelet. The training time is around 1 second per training run, on average, for both Cassandra and MongoDB, thanks to our context aware machine learning method.

**A. Cassandra Results**

For each of the above two anomaly scenarios, we have 20 normal runs and 20 runs with anomaly inserted for both the database systems. Each training run contains a complete run of the respective YCSB workload. The neural network is trained using the sampled data points from the 20 training runs and it extracts a representative wavelet for each workload. In the next step of training, we build the wavelet database by computing the DTW distance between the representative workload and the training wavelets from the 20 runs of each workload and storing the maximum distance, the workload type and the representative wavelet. The training time is around 1 second per training run, on average, for both Cassandra and MongoDB, thanks to our context aware machine learning method.

**A. Cassandra Results**

For each of the above two anomaly scenarios, we have 20 normal runs and 20 runs with anomaly inserted for both the database systems. Each training run contains a complete run of the respective YCSB workload. The neural network is trained using the sampled data points from the 20 training runs and it extracts a representative wavelet for each workload. In the next step of training, we build the wavelet database by computing the DTW distance between the representative workload and the training wavelets from the 20 runs of each workload and storing the maximum distance, the workload type and the representative wavelet. The training time is around 1 second per training run, on average, for both Cassandra and MongoDB, thanks to our context aware machine learning method.

**A. Cassandra Results**

For each of the above two anomaly scenarios, we have 20 normal runs and 20 runs with anomaly inserted for both the database systems. Each training run contains a complete run of the respective YCSB workload. The neural network is trained using the sampled data points from the 20 training runs and it extracts a representative wavelet for each workload. In the next step of training, we build the wavelet database by computing the DTW distance between the representative workload and the training wavelets from the 20 runs of each workload and storing the maximum distance, the workload type and the representative wavelet. The training time is around 1 second per training run, on average, for both Cassandra and MongoDB, thanks to our context aware machine learning method.
For creating test data for the detection of anomalies, we use 10 normal runs and 10 runs with anomaly inserted for each workload. Figure ?? shows the DTW distance from the representative run (black) that was computed for new normal runs not seen during training (light grey) and anomalous runs (grey), respectively, for a YCSB workload on Cassandra. It also shows the CPU utilization for the predicted, normal and anomalous scenarios. Figure ?? shows the same things, but for the resource interference anomaly. The disparity between the DTW distance in the normal and faulty runs is even more pronounced in this case.

B. MongoDB Results

Figure ?? shows the representative wavelet extracted for CPU utilization by the recurrent neural network (RNN) for a particular workload (workload-a) for MongoDB. Again we show 7 representative training runs in the figure and we show the extracted pattern in dotted line with a new run shown in solid lines at the end. We see that, in the case of MongoDB, there is more variation in CPU utilization than for Cassandra. However, there is a repeatable low-high-low utilization pattern for this workload, which is extracted successfully by the RNN.

Figure ?? shows the DTW distance computed for the new normal run, not previously seen in training (light grey) and anomalous runs (grey) from the representative run (black) for the YCSB workload on MongoDB. The figure also shows the CPU utilization for the predicted, normal and anomalous scenarios. Around time 65 seconds we inserted the disk delay for around 10 seconds. This anomaly is also captured by our method using the DTW distance.

C. Using User Feedback

Out of total 120 test scenarios of Cassandra and MongoDB, at first, we had 15 false positives (normal runs flagged as anomaly) and 10 false negatives (anomalous runs flagged as normal). As mentioned before, we buffer the runs triggered as anomalous until the administrator diagnoses the real problem. In case of false positives, we update our wavelet database with new normal trends and the treatment of false negatives are done using future detection of anomalies and creating representative databases for anomalous runs. After incorporating user feedback, we ran 20 more runs, half normal and half anomalous, for each of Cassandra and MongoDB. Out of these new test dataset of 160 runs (that included the 120 test runs that we have initially), our method is able to reduce the total number of false positives to 2 and false negatives to 0.

V. RELATED WORK

In this section, we briefly classify the existing techniques in anomaly detection and compare them with our work.

A. Theoretical Model-Based Approaches

Theoretical model-based approaches [?], [?], [?], [?] leverage theoretical concepts such as Queuing theory [?] to contrast observed system-behavior with theoretically expected behaviour, which may be based on unrealistic assumptions of system behaviour.

B. Expectation-Based Approaches

System programmers as well as system analysts can detect anomalies through expressing their expectations about the system’s communication structure, timing, and resource consump-
tion. Language based approaches include MACE [?], TLA+ [?], PCL [?], PSpec [?] and Pip [?]. PSpec [?] is a performance checking assertion language that allows system designers to verify their expectations about a wide range of performance properties. Performance analysis tools [?], [?], [?], [?], [?], [?] allow programmers to analyze the performance of a system to find sources of inefficiencies.

The limitation of these methods compared to our work is that they assume extensive knowledge of the user about the system's communication structure, timing, and resource consumption; anomalous patterns in modern large-scale systems may involve a combination of states and factors that cannot always be predefined.

C. Mining Operational Logs

DISTALYZER [?] provides a tool for developers to compare a set of baseline logs with normal performance to another set with anomalous performance. It categorizes logs into event logs and state logs. Xu et al. [?] leverage the schema of logging statements in the source code to parse log records. Fu et al. [?] use Finite State Automata to learn the flow of a normal execution from a baseline log set, and use it to detect anomalies in performance in a new log set. Similar to other off-line approaches [?], [?], [?], [?], [?], they use text mining techniques to convert unstructured text messages in log files to log keys, which requires expensive text processing. Yuan et al. [?] introduce techniques to augment logging information to enhance diagnosability.

Unlike these previous anomaly detection techniques, we are focused on recognizing patterns in resource monitoring data, which may incur less overhead than processing verbose text logs.

D. Context-Awareness in Anomaly Detection

Cohen et al. [?] correlate system metrics to high-level states, and leverage the model to find the root-cause of faults based on past failure cases within similar settings [?]. Similarly, Lao et al. [?] identify known problems by mapping traces of function calls to known high level problems and form a knowledge-base of association between symptoms and root-causes. Liblit et al. [?] samples traces from multiple instances of a program from multiple users to limit the overhead of monitoring on each user. They use classification techniques to predict correlation between traces and known bugs.

In recent, gray-box solutions [?], [?], part of the application context in which the anomaly occurs is captured and/or recovered as part of automatic monitoring or data analytics for anomaly detection in two main ways: via manual instrumentation or automatically, through static analysis of application code. The part of the high level context thus captured is then leveraged for root cause analysis of anomalies. For example in Zhao et al. [?] the application context is partly captured through static analysis and also recovered through automatic inferences based on data flow analysis for block identifiers in log records. The above-mentioned gray-box methods present several trade-offs such as: run-time overhead, storage overhead, post-processing overhead and the extent of manual user intervention. Our work reduces all types of overhead and needs trivial instrumentation.

Sambasivan et al. [?], Pip [?], PinPoint [?] infer context in terms of execution flow and diagnose anomalies by comparing the expected execution flow with the anomalous one. The common denominator of these approaches is detailed fine-grained tracing, which is provided by explicit code instrumentation or thorough distributed tracing enablers such as x-trace [?], Dapper [?] and Maggie [?]. Due to high monitoring resolution and causality information, trace-based anomaly detection techniques are more precise than regular log-based approaches. However, tracing generates a large amount of data which needs to be processed offline. Also, because of the high resolution data collection, they impose undue runtime overhead.

Sherlog [?] infers likely paths of execution from log messages to assist the operator in diagnosing the root-cause of failures. Like the above methods, SherLog is a postmortem error diagnosis tool. In contrast, just like our previous work [?] our resource monitoring work in this paper is designed for real-time anomaly detection.

VI. Conclusion

With a dramatic increase in scale and complexity of cloud infrastructure it is no longer possible for the human user to diagnose anomalies by searching through vast amounts of log and monitoring information with no "guiding map".

Our contribution in this paper is to bridge the gap between machine-centric black-box data mining anomaly detection solutions, and human-centric theoretical or language-based solutions. In the former, the machine attempts to process vast amounts of operational data with little or no user support. In the latter, the human is expected to know and predeclare all or most system and application patterns beforehand.

We believe that it is unreasonable to expect a reliability solution to work at either extreme of the automation spectrum in complex or large-scale distributed software for the cloud, such as Cassandra, and MongoDB. Instead, we propose a semantic-aware reliability solution that puts the human back in the reliability loop, while still working with existing automatic, legacy, fault-masking, fault-tolerance and logging solutions, and with minimal modifications to running software systems. Our key idea is to record resource monitoring data pre-labeled with the semantic application contexts they belong to. In this way, instead of poring over vast amounts of machine generated low level data, the human is relegated to tasks that humans are naturally good at performing i.e., inspecting interesting high level patterns and digging down into unusual patterns, that are already signalled by the machine learning runtime. Thus, the user can obtain information from the system, and give feedback to the system about a select few unusual patterns. This enables a synergy between human and machine into a best of both worlds solution, combining the strengths of both.
Our semantically-tagged model is based on RNN models, and allows association of resource consumption signals with the workload patterns they occur in. The model thus serves as a reference for both automatic runtime pattern classification and the user’s precise interpretation or diagnosis of a detected anomalous pattern.

We believe that our highly accurate and low overhead anomaly detection tool will form the basis for an effective real-time approach which synthesizes monitoring data into semantically rich patterns that convey meaning according to a learned model of the system per-context.

REFERENCES


[47] YCSB Core workloads, Available at: https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.