An anomaly detection approach for scale-out storage systems

Guthemberg Silvestre[1,2], Carla Sauvanaud[1,3], Mohamed Kaâniche[1,2], and Karama Kanoun[1,2]
[1]CNRS, LAAS, 7 Avenue du colonel Roche, F-31400 Toulouse, France
[2]Univ de Toulouse, LAAS, F-31400 Toulouse, France
[3]Univ de Toulouse, INSA de Toulouse, LAAS F-31400 Toulouse, France
Emails: {gdasilva,csauvana,mohamed.kaaniche,karama.kanoun}@laas.fr

***Abstract*—Scale-out storage systems (SoSS) have become increasingly important for meeting availability requirements of web services in cloud platforms. To enhance data availability, SoSS rely on a variety of built-in fault-tolerant mechanisms, including replication, redundant network topologies, advanced request scheduling, and other failover techniques. However, performance issues in cloud services still remain one of the main causes of discontentment among their tenants. In this paper, we propose an anomaly detection approach for SoSS that predicts cloud anomalies caused by memory and network faults. To evaluate our prediction model, we built a testbed simulating a virtual data center using VMware. Experimental results confirm that the injected faults are likely to undermine the data availability in SoSS. They suggest that although unsupervised learning has been the most common method for anomaly detection, a supervised-based implementation of the same model reduces the false positive rate by roughly 10%. Our analysis also points out that probing SoSS-specific monitoring data at the VM-level contributes to improve the anomaly prediction efficiency.**

I. Introduction

Scale-out storage systems (SoSS) have become increasingly important to provide dependable web applications in cloud platforms. In a cloud-friendly data center environment, back-end data stores of multiple users, or tenants, share resources in order to provide horizontal scalability. Per-tenant data store is partitioned and replicated throughout different storage service instances for improving its scalability and availability. At the virtual machine (VM) level, a service instance can also share virtualized resources among different data store replicas to improve resource allocation.These SoSS allow cloud providers to enhance the platform utilization and to reduce costs. Popular SoSS include key-value stores, document stores, distributed relational databases, and block storage. So far, they have been successfully used for numerous web applications, such as threaded web conversations and posts, distributed monitoring systems, big data stores, photo tagging, user status updates, sessions and profiles.

Yet performance issues in distributed systems are still major causes of discontent among cloud tenants [17], [26]. The root cause of unwanted performance variations of SoSS in cloud environments may depend on a wide range of uncorrelated issues, including systems permanent and transient faults, configuration errors, software bugs, limping hardware [9], to name a few. These performance issues can be grouped into two broad classes: failures and resource allocation problems. Failures in large SoSS are likely to be common. Although their failover mechanisms handle fail-stop failures successfully, many other failures may remain unnoticed by fault-tolerant mechanisms. For instance, Do *et al.* [9] found that a single limping network interface can cause a three orders of magnitude execution slowdown in distributed key-value data stores. Similarly, as many consolidated VMs must compete for shared resources in a single physical host, efficient resource allocation in cloud platforms becomes quite challenging. Actually, this may result in contentions of hosts resources such as main memory and network bandwidth leading to increasing completion times of operations [27], [15]. Therefore, we believe that cloud providers must be able to uncover these performance issues in order to enhance the dependability of SoSS.

Anomaly detection has been a widely used technique to automatically identify performance issues in large-scale distributed systems [6]. Most of the anomaly detection approaches for distributed systems are based on a unsupervised learning method. To boost their performance, they rely on data mining techniques [20], [14], [19], [13]. Yet prediction efficiency remains the main drawback of this method [21], as it is statistically difficult to ascertain the quality of inferences drawn from the predictions of unsupervised learning algorithms. Moreover, to the best of our knowledge, there is not yet a precise analysis of the prediction efficiency of anomaly detection approaches in SoSS. In particular, none has analysed the predictability of anomalies in SoSS that use replication and load balancing to enhance data availability.

This work introduces an anomaly detection approach for SoSS. We are particularly interested in identifying anomalous VMs with high prediction efficiency. To evaluate our anomaly detection approach, we built a testbed simulating a virtual data center using VMware. In this testbed, we deployed a cluster of VMs to run MongoDB [2], the most popular document store nowadays[1]. We injected memory and network faults in VMs to provoke cloud anomalies, then we performed a precise analysis of the predictability of these anomalies. Our analysis was threefold. First, we assessed the impact of cloud anomalies on a MongoDB cluster. Then, we compared the efficiency of our prediction model implemented as supervised and unsupervised learning methods. Finally, we evaluated the quality of different monitoring data with respect to the probing sources, particularly whether it is worth monitoring directly VMs operating systems and SoSS utilization counters.

Experimental results suggest that the injected faults are likely to undermine the data availability of a MongoDB cluster

[1]http://db-engines.com/en/ranking

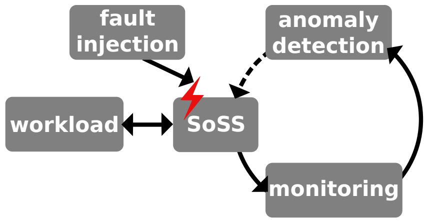


Fig. 1: Methodology overview and the functional blocks.

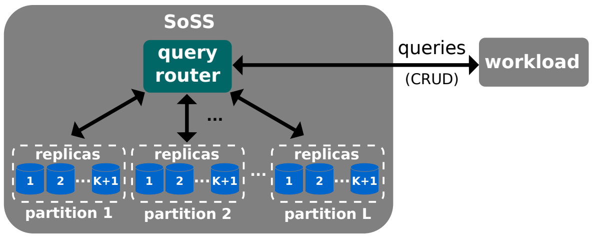


Fig. 2: SoSS architecture and workload.

regardless of its built-in fault-tolerant mechanisms, notably replication and load balancing of queries. They also highlight that a supervised learning-based implementation of our prediction model performs much better than an unsupervised one in predicting these anomalies, reducing by roughly 10% the false positive rate. Our findings point out that collecting SoSS-specific monitoring data directly from VMs provides a worthwhile improvement in the prediction efficiency of our anomaly detection approach.

II. Anomaly Detection Approach

In this section, we describe our approach to enhance the capacity of SoSS to detct anomalies. In particular, we define our anomaly prediction model that classifies VMs in anomalous or normal based on the monitoring data of SoSS. Figure 1 provides a high-level overview of our approach. It introduces the five main functional blocks of our approach, namely SoSS, workload, fault injection, monitoring, and anomaly detection. In the remaining of the section, we detail each one of these blocks.

A. SoSS

SoSS is the subject of our anomaly detection study. Figure 2 highlights the main components a *SoSS* and the interactions with the workload. To enhance data availability for cloud services, *SoSS* provides scalable and fault-tolerant data storage. Such systems offer scalable storage service by allowing cloud operators to evenly split data in L partitions or shards across a cluster of VMs. To increase the storage capacity of the cluster, operators may add nodes as the system run. Partition replication is the most common mechanism to enforce data durability and availability in *SoSS*. Each partition has a predefined number of replicas K+1, which allows the system to tolerate up to K VM failures. Copies of the same partition form the so-called replica set. Depending on the scheme to maintain replicas, replication is divided in two broad groups: primary/secondary and multi-primary schemes. Essentially they differ on how requests that modify data are handled. Our approach takes into account both schemes. In addition, we assume that *SoSS* rely on load balancing throughout replicas in order to limit the impact of cloud anomalies on a cluster.

B. Workload

Workload functional block loads a *SoSS* cluster with data and runs the workload for evaluation purposes. Through this block, we can define workload settings, such as document size distribution, the rate of query per second, and the distribution popularity of documents. As depicted in Figure 2, the *workload* allows us to generate four types of queries, namely create, read, update, and delete (CRUD) queries.

C. Fault Injection

Fault injection allows us to emulate malfunctioning and errors events in cloud environments to cause anomalous behaviours in the *SoSS*. We investigate the impact of faults that are hard to identify and degrade significantly the performance of our *SoSS*. Our goal is to provide a collection of faulty behaviours that are likely to occur in cloud environments and that existing fault-tolerant mechanisms may fail to deal with. In this work, we focus on the following two categories of faults.

Network faults. Communication issues in distributed systems are common, and they have a significant impact on the performance of SoSS. To analyse their impact, we inject three types of network faults, namely packet loss, network latency, and limping network. Packet loss and network latency allowed us to emulate common interconnection issues of cloud platforms, such as network partition in data center networks. Limping network is intended to reproduce networks' anomalous behaviours previously observed by Do *et al.* [9]. This fault emulates a limping network interface, whose actual transfer rate does not comply with the manufacturer's specification.

Memory faults. As the price of main memory continues to drop, distributed data store operators have been encouraged to fit data to main memory in order to improve performance [25]. Hence, faults in main memory have become increasingly harmful to these systems. We intend to emulate faults related to misuse of memory. At the VM level, this kind of fault allows us to make arbitrary amounts of main memory inaccessible to a SoSS. The aim is to provoke an abnormal functioning of a SoSS due to an unexpected utilisation of main memory. Consequently, there will be less main memory available and more unwanted, costly disk I/O. A typical example of this fault is a VM running out of memory due to a misconfiguration, memory leaking, overloading, or an unbalanced resource allocation.

D. Monitoring

Monitoring functional block allows us to collect data that quantifies the utilization of the cloud resources for the *SoSS*. To collect our datasets, we consider the monitoring system depicted in Figure 3. The monitoring system (i) collects measurements from the SoSS through probing agents; (ii) gathers and stores these measurements in a `rrd` file; and (iii) processes these files periodically to create our raw dataset in a relational database. To characterize the state of a VM, this system collects data from the monitoring counters of

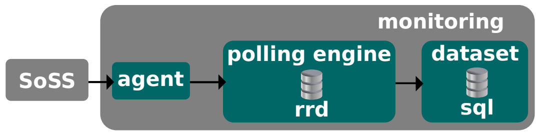


Fig. 3: Monitoring system and datasets.

the storage service, like number of served queries, and from the cloud infrastructure, such as CPU and memory usage. Probing agents collect SoSS-specific monitoring data directly from the VMs, and generic activity monitoring data from the hypervisors or VMs.

E. Anomaly Detection

This is the main functional block of this study. We briefly introduce the main statistical learning methods that are implemented and compared in this work. Then we describe the prediction model. Finally, we present the metrics to evaluate the efficiency of our anomaly detection approach.

1) Statistical learning methods: Statistical learning is about learning from seen data in order to predict unseen data with minimal error. Data comprises measurements represented by a feature vector $\mathbf{x}$ with a fixed number of dimensions d ($\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^d$) from the input space $\mathcal{X}$. There are two ways of learning from data: supervised and unsupervised learning.

In supervised learning, each measurement or input is coupled with a y, a label, from the output space $\mathcal{Y}$. To learn, we have N pairs $(\mathbf{x}, y)$ drawn *independent and identically distributed* (i.i.d.) from a fixed but unknown joint probability density function $Pr(X, Y)$. This is true for both training and testing datasets. For instance, we consider the training dataset $S = \{\mathbf{x}_i, y_i\}_{i=1}^N$ of N pairs $(\mathbf{x}, y)$. From this dataset, the supervised learning algorithm searches for a function $f : \mathcal{X} \to \mathbb{R}$ in a fixed function class $\mathcal{F}$. State-of-the-art algorithms, such as *support vector machines* (SVM) [18] or *Adaboost* [11], aim to find $f^\star$ in $\mathcal{F}$ with the lowest empirical risk defined as:

$$f^\star \in \arg\min_{f \in \mathcal{F}} \mathtt{r}_{emp}(f)$$

where $\mathtt{r}_{emp}(f) = \frac{1}{N} \sum_{i=1}^N I_{\{f(\mathbf{x}) \neq y_i\}}$ is computed over the training set, and $I_{\{.\}}$ is the indicator function which returns 1 if the predicate $\{.\}$ is true and 0 otherwise.

In unsupervised learning, we have N unlabelled samples $(x_1, x_2, \dots x_N)$ of a random d-vector X having probability density function $Pr(X)$. Unlike supervised learning, there is no outputs to learn from. Instead, we are interested in inferring the properties of function $Pr(X)$. This allows us to have insights into how the data is organized or clustered.

The unsupervised learning method allows us to make fewer assumptions about data, since they do not require labelled training datasets. To boost their efficiency, they rely on data mining techniques [20], [14], [19], [13]. Yet prediction performance remains the main drawback of these approaches [21], as it is statistically difficult to ascertain the quality of inferences drawn from the predictions of unsupervised learning method. With supervised learning, however, there is a clear measure of success, that is quantified by the expected loss over the joint probability density function $Pr(X, Y)$ [18].

2) Anomaly prediction model: The goal of our prediction model is to detect anomalies in VMs that are likely to undermine the performance of SoSS. We intend to compare the prediction efficiency of two implementations of this model, as a supervised and as a unsupervised learning method. In our model, $\mathbf{x}$ represents a VM. The model classifies VMs in anomalous or normal with respect to their current state. The output of our model is defined as follows.

$$\hat{G}(x) = \begin{cases} 1 & \text{if a VM is behaving anomalously.} \\ -1 & \text{if a VM is behaving normally.} \end{cases}$$

This model makes predictions based on monitoring dataset, described in Subsection II-D. Monitoring dataset is preprocessed according to the statistical learning method to be performed. With supervised learning, we add labels to the training dataset. After preprocessing monitoring datasets, we set to 1 labels of input measurements of faulty VMs (Subsection II-C) in the training dataset. Hence, each line of our datasets for anomaly detection has d inputs and one output label that indicates if a VM behaves anomalously or not. This procedure is slightly different for the unsupervised learning method. Since it does not require labelled data to learn, its training dataset has d inputs with data from VMs where faults were not injected and behaved normally. To test or validate our model as an unsupervised learning method, the preprocessing procedure is the same as that for supervised learning one. We detail the use of our dataset for predictions in Subsection IV-B.

3) Prediction efficiency metric: We evaluate the efficiency of our binary classifier with respect to two different statistical learning methods using *average precision*. *Average precision* is a single-value efficiency metric widely used in Information Retrieval [18]. We chose this metric because it combines two other key efficiency metrics: *precision* and *recall*. Precision is the rate of anomalous events detected successfully over the total number of events. Recall measures the rate of anomalous events detected successfully over the total number of ground truth anomalous events. To find the *average precision* of a binary classifier, we compute its *precision* (p) averaged across all values of *recall* (r) between 0 and 1. It is defined as follows:

$$\textit{average precision} = \int_0^1 \mathrm{p(r)}\,\mathrm{dr} \tag{1}$$

Actually, the *average precision* corresponds to the area under the precision-recall curve. Finally, we define the *prediction error* of a learning model as follows:

$$\textit{prediction error} = 1 - \textit{average precision} \tag{2}$$

III. Experimental Set-up

This section details the experimental set-up to measure the efficiency of our anomaly detection approach.

A. Testbed Settings for Simulating a Virtual Data Center

Our experimental testbed consists of two Dell PowerEdge R620 hosts, namely target and experimentation hosts. Each host has two Xeon E5-2660 2.2 GHz processors, 64 GB of memory, and two 130 GB SATA disks. The hosts were connected by Gbit Ethernet. These resources are shared among the VMs and services running on each host. We chose VMware as the virtualization technology and ESXi 5.1.0 as hypervisor. Figure 4 depicts our virtual data center for experimentations, highlighting how we consolidated VMs across hosts.

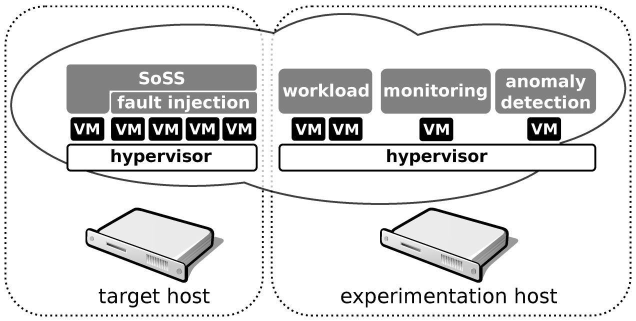


Fig. 4: Experimental testbed.

The target host runs our SoSS, monitoring agents, and our fault injection tools whereas experimentation host runs everything else, notably the workload, monitoring system, and the prototype of our anomaly detection approach. By grouping our SoSS cluster along with the fault injection tools in a single host, we aim to reduce any unwanted noise or load on the monitoring data of storage nodes.

VMs were consolidated according to their profiles. Profiles are loosely based on functional block of our approach, detailed in Section II. They essentially differ in main memory, disk, and, particularly, network capacity. To prevent a misleading network utilization, we connected storage nodes to 100Mbps virtual network. The workload functional block has two VMs with identical settings with larger amounts of main memory, which allows us to control workload settings. The VM for monitoring requires a bigger disk, just like anomaly detection's VM, that runs statistical learning algorithms over big amounts of monitoring data.

B. SoSS and Workload Settings

As SoSS, we chose MongoDB (2.4.8 release) [2]. We consolidated in our experimental testbed a MongoDB cluster following the *SoSS* definitions of Subsection II-A. In our cluster, we set the replication factor K to 1 and the number of partitions L to 2. We consolidate five VMs to deploy our cluster, query router and four document stores. Data is evenly distributed by a query router of MongoDB throughout the two partitions using the hash code of documents' keys. In MongoDB, there is a single primary replica on each replica set. Replicas regularly exchange heartbeat messages to elect the primary copies as failures occur. We used the default timeout setting of 10 seconds for primary replica election. This set-up forms a small but resilient, fault-tolerant SoSS, which allows us to measure the impact of different faulty scenarios.

We investigate the performance of MongoDB using the Yahoo! Cloud Serving Benchmark (YCSB) [8], a workload generator and benchmark tool for SoSS. In a preliminary setup, we use YCSB to load documents whose size varies from 1KB to 3KB. Once the database is loaded, we set-up YCSB to generate a mostly-read workload. The popularity of documents follows a Zipf-like distribution. We evaluated the MongoDB performance in serving an average throughput of 5000 queries per second. YCSB provides two key metrics to evaluate the performance of SoSS, average throughput rate and 99^{th} percentile latency. In a fault-free scenario, our MongoDB cluster serves the expected 5000 queries per second with a 99^{th} percentile latency smaller than 12 milliseconds.

C. Fault Injection Campaigns

We developed injection scripts to provoke cloud anomalies and to investigate the impact of a faulty cloud environment in our MongoDB cluster. These scripts allow us to inject network and memory faults detailed in Subsection II-C. We implemented network faults using Dummynet [5], a widely used link emulator. For memory faults, we implemented Python scripts that allocated arbitrary amounts of main memory. Table I summarizes the main characteristics of our fault injection campaigns. We can assign 10 intensity values or levels to a fault. Although the range and distribution of fault intensities had been chosen arbitrarily, they provide a fair insight about how these faults could lead to anomalies of cloud environments. Our scripts have three parameters to define a single injection: type of fault, intensity, and duration. For each VM hosting a MongoDB document store, we injected a series of faults including all types and intensities. Between two injections, script procedures ensure that the MongoDB cluster recovered completely from faults. We did not inject faults or different fault intensities concurrently.

D. Monitoring System and Datasets for Analysis

Our monitoring system, whose design is described in Subsection II-D, was built on top of Ganglia [22], a scalable distributed monitoring system for high performance computing systems. We deployed a Ganglia agent on each hypervisor and VM. Every 30 seconds, they collect data about the utilization of the cloud resources by the SoSS. These measurements, including anomaly injection information, are organized by VM and stored in a relational database every minute. To assess the effect of different probing sources on anomaly detection of VMs, measurements are organized in three source groups, namely A, B, and C, as described in Table II.

TABLE II: Probing sources for monitoring.

Label	Source	Agent deployment	Measurements
A	SoSS	VM	16
B	TCP network layer	VM	21
C	`systat` of VMs or hypervisor	VM or hypervisor	23

The behaviour of a VM of a SoSS cluster is characterized by these 60 measurements (*i.e.*, $d = 60$). Monitoring sources have a sightly similar number of measurements. They allow us to evaluate the trade-off between collecting generic VM-level data and collecting service-level data. Sources A and

TABLE I: Fault injection campaigns.

Fault type	Duration (days)	Unit	Intensity levels									
			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Packet loss	3	%	0.8	1.6	2.4	3.2	4.0	4.8	5.6	6.4	7.2	8.0
Network latency	3	ms.	8	16	24	32	40	48	56	64	72	80
Limping network	3	Mbps	32	10	3.2	1.0	0.32	0.1	0.032	0.01	0.0032	0.001
Misuse of memory	3	%	70	73	76	79	82	85	88	91	94	97

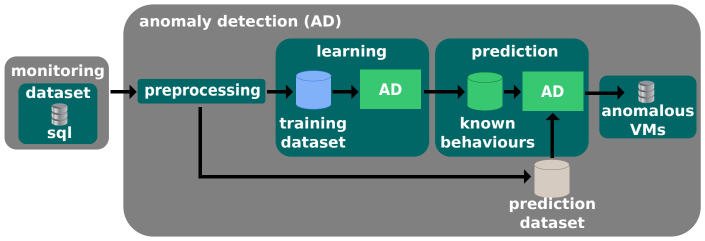


Fig. 5: Anomaly detection module for SoSS.

B require a probing agent running inside the VM, whereas measurements from source C come from an agent probing directly the hypervisor monitoring counter. Among the probing sources, only source A is specific to the target SoSS. The list of all 60 measurements, so-called anomaly detection inputs, is available online[2]. Overall, our dataset for analysis has 50,000 samples based on VMs, including 15,000 one-minute anomalous samples.

E. Anomaly Detection Implementation

The prototype of our prediction model is based on *support vector machine* (SVM) learning algorithm [18]. Our anomaly detection module was implemented using Scikit-learn library [23].

We designed and implemented a anomaly detection (AD) module to identify anomalies in SoSS, depicted in Figure 5. The AD has two operating phases: (i) learning and (ii) prediction. Each phase has its own dataset collected by the monitoring system. In the learning phase, monitoring data is preprocessed to generate the training dataset. Then this training dataset is fed to the AD module, whose functioning depends on the implemented statistical learning method. If the AD is based on a supervised learning method, it will identify normal and anomalous patterns, otherwise, the AD is based on an unsupervised learning method, it characterizes normal patterns only. Once the learning phase has been accomplished, AD can use its learning module in a prediction phase, as indicated in the right-hand side of Figure 5. In this phase, inputs come directly from measurements of the monitoring system, that permit classifying the behaviour of a VM into anomalous or normal.

IV. Evaluation Results

In this section, we analyse the impact of the injected faults in MongoDB cluster, we compare the prediction efficiency of these faults for two implementations of our model, and we assess the quality of information from the different sources of monitoring data for anomaly predictions.

[2]http://homepages.laas.fr/gdasilva/datasets/oltp_mongodb.info

A. Measuring the impact of faults on SoSS

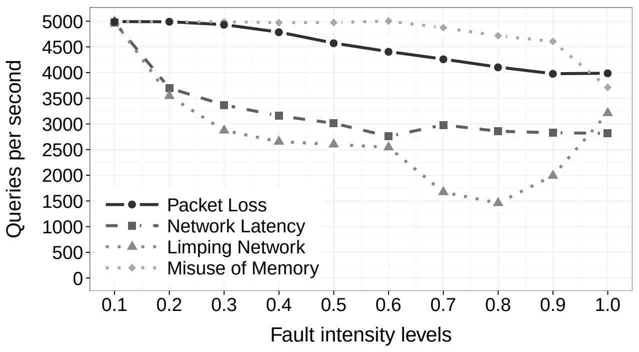
Figures 6a and 6b show the impact of our fault injection campaigns, detailed in Subsection III-C. From our experimental results, we highlight the following observations.

- **Impact on the average throughput rate.** Our MongoDB cluster served 5000 queries per second before injecting faults. Under faults, the average throughput could be reduced by more than two thirds as a single storage node has a limping network interface [9]. The impact of increasing network latency fault was considerably high. According to its intensity level, the average throughput ranged from 4976 to 2817 queries per second. We also observed that the impact of packet loss on the average throughput increased smoothly. It remained stable as the rate of packet loss was higher than 7.2%. Considering the average throughput rate, MongoDB seems to be very resilient against memory faults. For instance, for a memory fault intensity as high as 72% of all main memory (i.e., only 28% of main memory in available for the MongoDB instance), there was no impact on the average performance. Surprisingly, we started observing a performance degradation of its average throughput when less than a quarter of the main memory was available for all other running processes of the VM.
- **Impact on the 99^{th} percentile latency.** In a fault-free scenario, we observed a 99^{th} percentile latency of 12 milliseconds. When we inject a limping network fault, this metric reached a peak of 389 milliseconds, in other words, a 32-fold increase. Similar to the impact on average throughput, limping network seems to be the worst type of fault. However, as we shrank the available bandwidth to values lower than 100Kbit/s (a limping network fault with an intensity of 0.6), MongoDB identifies the faulty node and overcomes the problem by sending queries to another replica. It confirms how important network resources are for this kind of system.

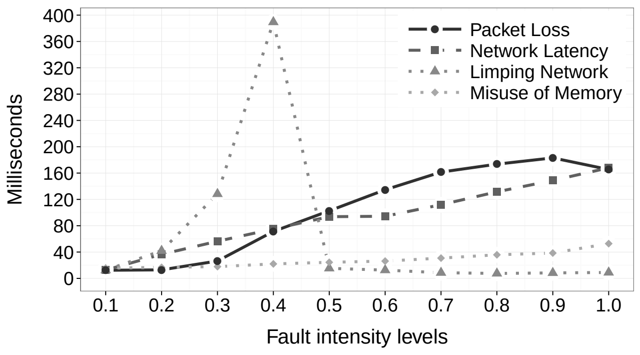
Overall, all faults undermined the performance of our MongoDB cluster, despite the load balancing of queries and management of the liveness of replicas. We observed these faults had a higher impact on the 99^{th} percentile latency.

B. Prediction about Anomalies in Cloud Environments

The evaluation of our prediction model has three operating phases: training, cross-validation, and prediction. While training and prediction correspond exactly to the two phases of our anomaly detection framework, detailed in Subsection III-E, cross-validation actually is a preliminary phase that helps us



(a) Average throughput rate.



(b) 99^{th} percentile latency.

Fig. 6: The impact of cloud anomalies caused by network and memory faults on the data availability of MongoDB.

to design our prediction model, e.g. to evaluate inputs and to select a learning method. We evaluate the efficiency of our anomaly detection framework using the dataset described in Subsection II-D. We split our dataset of 50,000 samples (described in Subsection III-D) in three parts: 60% for training dataset, 20% for cross-validation, and the remaining 20% for prediction. We remind that our model predicts a positive value, 1, whenever an anomalous VM is detected, otherwise -1.

C. Cross-validation with Supervised and Unsupervised Learning Approaches

Cross-validation is an essential procedure to evaluate the quality of a learning model. In particular, this evaluation step allows us to verify if our learning model is overfitting or underfitting. Overfitting is a problem in a statistical learning method that prevents a model to generalize, in other words, the model data fitting is too specific. Underfitting has an opposite effect. If a model suffers from underfitting, it performs poorly, resulting in a high number of wrong predictions. In both cases, the model must be improved.

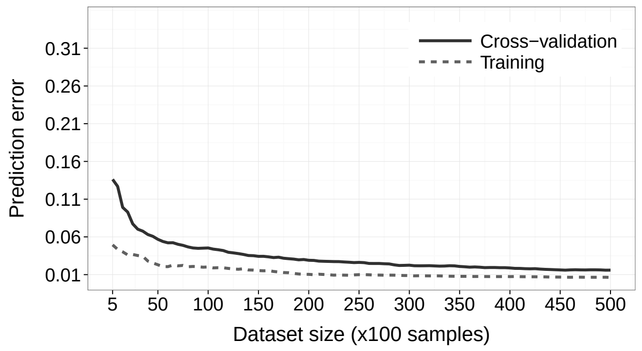
One way to verify overfitting and underfitting is by plotting prediction error curves of the different learning methods. These curves help us to choose the "best" learning method. In our evaluation, the error curves are based on the efficiency metric defined in Section II-E, *average precision* (Equation 1) and *prediction error* (Equation 2). Moreover, we plot the error curves as a function of the sample size to verify if our dataset has enough information for accurate predictions. Figure 7 depicts the error curves of our anomaly prediction model using an unsupervised learning and supervised learning methods. Plots have 20 points each. A point corresponds to the *prediction error* with a portion of the datasets. For instance, the evaluation of the first point takes into account one twentieth of training dataset and one twentieth of cross-validation dataset. As the x-axis values increase, datasets become progressively bigger. The twentieth point contains the entire training and cross-validation datasets. To compute the *prediction error* of a point, we train our model with its respective training dataset size and we measure error of prediction about its own training dataset and the cross-validation dataset. These plots allow us to draw the following conclusions about the models.

- **None of them was underfitting**. As the sample size of training and cross-validation datasets increases, the gap between the prediction error curves of both learning methods decrease significantly and then remains unchanged. This means that additional training data is unlike to help us to reduce prediction error. Our datasets seem to be large enough for training purposes of our anomaly detection model.
- **Our model was overfitting when implemented as an unsupervised learning method**. Our model performed predictions about anomalies with high *prediction error* as we used an unsupervised learning-based implementation. With this approach, *prediction errors* were higher than 0.2. When we implemented our model using a supervised learning method, however, it did not suffer from overfitting. Actually, we were able to reduce *prediction error* of our model to roughly 0.02. This result shows that the supervised-based implementation of our model generalizes properly and performs prediction with high efficiency.

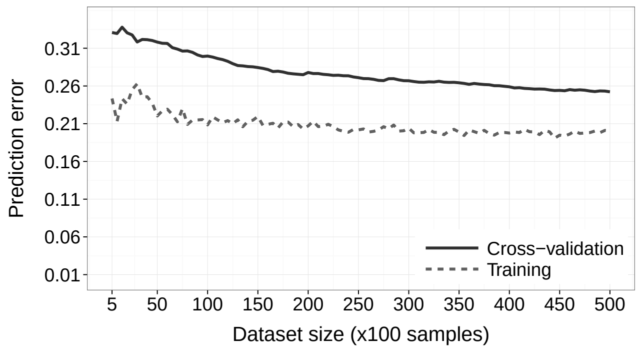
To provide a wider view of the efficiency of our model based on different statistical learning methods, we compute other prediction efficiency metrics. Figure 8 shows the false positive and true negative rates of supervised and unsupervised methods. It shows that a supervised learning method is highly efficient in our context. For instance, for this set of experiments, it reduces the number of false positives, or VMs wrongly classified as anomalous, from 0.11 using an unsupervised learning method to almost 0.01 using a supervised learning method. The smaller is the false positive rate, the easier is to use predictions to trigger recovery procedures accurately. Finally, Table III summarizes other performance metrics during our cross-validation evaluation. This table shows efficiency values using the entire training and cross-validation datasets. It allows us to compare the *average precision*, our main metric, to other metrics of prediction efficiency. These results suggest that all efficiency metrics are consistent and confirm the superiority of the supervised learning method.

D. Measuring the Contribution of Different Monitoring Sources to Prediction Efficiency

We computed the *average precision* for all combinations of our three sources of data monitoring, detailed in Subsec-



(a) Error curve for supervised learning model.



(b) Error curve for unsupervised learning model.

Fig. 7: The impact of network and memory faults on the data availability of SoSS.

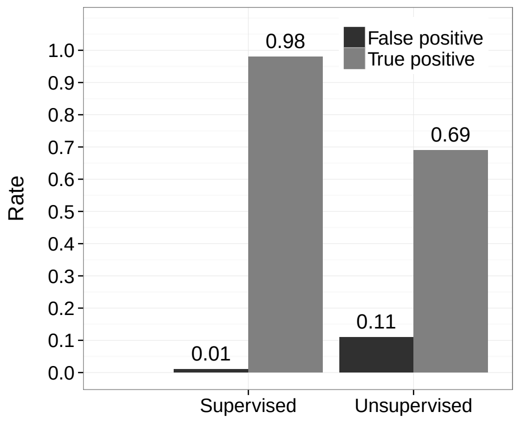


Fig. 8: False and true positive rates for two learning methods.

TABLE III: Comparing efficiency metrics for learning methods.

Metric	Learning method Supervised	Unsupervised
Average precision	0.98	0.75
AUC	0.98	0.79
Accuracy	0.99	0.83
Positive precision	0.98	0.72
Negative precision	0.99	0.87
Positive recall	0.98	0.68
Negative recall	0.99	0.69

tion II-D. Our goal was to assess the quality of information provided by each monitoring source, and how they contribute to provide high prediction efficiency. To achieve this goal, we used the supervised learning version of our prediction model, and we considered the training and the prediction datasets. We selected inputs for training our model according to the monitoring sources, namely A, B, and C. Figure 9 presents the results for different combinations of monitoring sources. If we consider only the monitoring data that comes from source C, which includes common VM-level measurements, we achieve a high prediction efficiency of 0.95. We increase the model efficiency by 0.03 when we add monitoring information from TCP-level or data storage usage sources. As any increase in efficiency contributes to the improvement of SoSS dependability, we conclude that including monitoring data from probes installed inside the VMs is worthwhile.

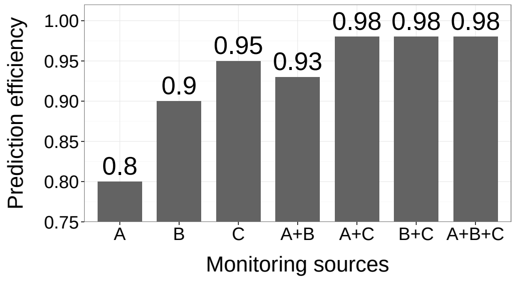


Fig. 9: Contribution of different monitoring sources to prediction efficiency.

V. Related Work

Fault tolerance in SoSS. Distributed systems use replication and advanced request scheduling to improve data availability. Ficus [16] and Bayou [24] are data storage that rely on replication to ensure data availability against fail-stop failures, but they are not able to deal with transient cloud failures. Skute [4] provides an adaptive replication scheme that mitigate the impact of cloud failures. However, it does not provide mechanisms to ensure high data availability, such as high throughput and bounded latency. Popular SoSS, like MongoDB [2], CouchDB [1] and Redis [3], offer high data availability using an eventual consistency replica system and enhanced main memory data structures [28]. But, our study showed that cloud failures, like packet loss and main memory faults, can undermine considerably their data availability.

Cake [30] offers a scheduling scheme to enforce high-level data availability requirements for end users. Similarly, PARDA [15] enforces proportional-share fairness in the access of data in a storage cluster. Since neither Cake nor PARDA are designed to identify faulty VMs, our work is complementary to theirs. Eriksson *et al.* [10] provide a routing framework that helps cloud operators to mitigate the impact of network failures. Alerts from our approach can contribute to enhance the assessment of network outage risk of their framework.

Anomaly detection in SoSS. Prediction models for anomaly detection are commonly implemented based on an *unsupervised learning method*. Liang *et al.* [20], and Gujrati *et al.* [14] as well, provide prediction models based on event logs of supercomputers to identify platform-wide anomalies, whereas we are interested in detecting anomalous VMs based on common data center monitoring data. Chen *et al.* [7] proposes an anomaly detection approach for large-scale systems that improves the prediction efficiency of an entropy-based information theory technique by performing a principal component analysis (PCA) of system inputs. However, this introduces computational overhead that undermines its scalability and causes a slowdown in anomaly predictions. While we focus on detecting a small set of cloud anomalies with high prediction efficiency, Lan *et al.* [19] provides a general-purpose anomaly detection approach that strongly relies on input selection to enhance prediction efficiency. Similarly, Guan and Fu [13] performs inputs extraction based on PCA to identify the most relevant inputs for anomaly detection. Yet, our results confirm that an unsupervised-based approaches undermine the prediction efficiency of cloud anomalies.

Guan *et al.* [12] implement a probabilistic prediction model based on a *supervised learning method*. Although their model allows us to compare the dependability of virtualized and non-virtualized cloud systems, their results suggest that their model suffers from poor prediction efficiency when it is used to predict cloud anomalies. Tan *et al.* [29] propose general-purpose prediction model to prevent cloud anomalies. Their *supervised learning*-based model combines 2-dependent Markov chain model with the tree-augmented Bayesian networks. But, the authors did not provide information about the prediction efficiency and the capacity of generalize of their approach.

VI. Conclusion

Web services rely on SoSS to provide content with high availability. However, we showed that cloud anomalies can undermine the capacity of these systems to enforce data availability requirements. In this work, we presented an anomaly detection approach to enhance the dependability of SoSS through predictions of cloud anomalies. Our results show that our approach can efficiently identify anomalous VMs. Although anomaly detection in distributed systems is commonly implemented based on an unsupervised learning method, we showed that a supervised learning-based implementation of the same prediction model reduces the false positive rate by 10%. We also showed that collecting more specific probing sources at the VM level improves the detection of anomalous VMs in SoSS. As future research, we plan to extend our approach based on our preliminary findings in order to evaluate different SoSS, under different workloads and a large number of faults.

Acknowledgments

This research is partially funded by the project Secured Virtual Cloud (SVC) of the French program Investissements d'Avenir on Cloud Computing.

References

[1] Apache couchdb. http://couchdb.apache.org/.

[2] Mongodb. http://www.mongodb.org/.

[3] Redis. http://redis.io/.

[4] N. Bonvin, T. G. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *SoCC*, 2010.

[5] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM*, 2010.

[6] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM CSUR*, 2009.

[7] H. Chen, G. Jiang, and K. Yoshihira. Failure detection in large-scale internet services by principal subspace mapping. *TKDE*, 2007.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[9] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *SoCC*, 2013.

[10] B. Eriksson, R. Durairajan, and P. Barford. Riskroute: a framework for mitigating network outage threats. In *CoNEXT*, 2013.

[11] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT*, 1995.

[12] Q. Guan, C.-C. Chiu, and S. Fu. Cda: A cloud dependability analysis framework for characterizing system dependability in cloud computing infrastructures. In *PRDC*, 2012.

[13] Q. Guan and S. Fu. Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures. In *SRDS*, 2013.

[14] P. Gujrati, Y. Li, Z. Lan, R. Thakur, and J. White. A meta-learning failure predictor for blue gene/l systems. In *ICPP 2007*, 2007.

[15] A. Gulati, I. Ahmad, C. A. Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, 2009.

[16] R. G. Guy, J. S. Heidemann, W.-K. Mak, T. W. Page Jr, G. J. Popek, D. Rothmeier, et al. Implementation of the ficus replicated file system. In *USENIX*, 1990.

[17] J. Hamilton. The cost of latency, 2009.

[18] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[19] Z. Lan, Z. Zheng, and Y. Li. Toward automated anomaly identification in large-scale systems. *TPDS*, 2010.

[20] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *DSN*, 2006.

[21] B. C. Love. Comparing supervised and unsupervised category learning. *Psychonomic Bulletin & Review*, 2002.

[22] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 2004.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python . *JMLR*, 2011.

[24] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: replicated database services for world-wide applications. In *ACM SIGOPS European workshop: Systems support for worldwide applications*, 1996.

[25] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. In *VLDB*, 2012.

[26] E. Schurman and J. Brutlag. The user and the business impact of server delays, additional bytes, and http chunking in web search, 2009.

[27] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, 2012.

[28] M. Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 2010.

[29] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *ICDCS*, 2012.

[30] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: enabling high-level slos on shared storage systems. In *SoCC*, 2012.