Anomaly Detection and Diagnosis for Cloud services: Practical experiments and lessons learned

Carla Sauvanaud, Mohamed Kaâniche, Karama Kanoun, Kahina Lazri, Guthemberg Silvestre

▶ **To cite this version:**

Carla Sauvanaud, Mohamed Kaâniche, Karama Kanoun, Kahina Lazri, Guthemberg Silvestre. Anomaly Detection and Diagnosis for Cloud services: Practical experiments and lessons learned. Journal of Systems and Software, 2018, 139, pp.84-106. 10.1016/j.jss.2018.01.039 . hal-01864357

HAL Id: hal-01864357

https://laas.hal.science/hal-01864357

Submitted on 29 Aug 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Anomaly Detection and Diagnosis for Cloud services: Practical experiments and lessons learned

Carla Sauvanaud[a], Mohamed Kaâniche[a], Karama Kanoun[a], Kahina Lazri[b], Guthemberg Da Silva Silvestre[c]

[a]*LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France*
[b]*Orange Labs, 38 rue du General Leclerc, 92130 Issy-Les-Moulineaux, France*
[c]*ENAC, 7 avenue Edouard Belin, CS 54005, 31055 Toulouse Cedex 4, France*
***Emails**: firstname.name@laas.fr, firstname.name@orange.com, firstname.name@enac.fr*

Abstract

The dependability of cloud computing services is a major concern of cloud providers. In particular, anomaly detection techniques are crucial to detect anomalous service behaviors that may lead to the violation of service level agreements (SLAs) drawn with users. This paper describes an anomaly detection system (ADS) designed to detect errors related to the erroneous behavior of the service, and SLA violations in cloud services. One major objective is to help providers to diagnose the anomalous virtual machines (VMs) on which a service is deployed as well as the type of error associated to the anomaly. Our ADS includes a system monitoring entity that collects software counters characterizing the cloud service, as well as a detection entity based on machine learning models. Additionally, a fault injection entity is integrated into the ADS for the training the machine learning models. This entity is also used to validate the ADS and to assess its anomaly detection and diagnosis performance. We validated our ADS with two case studies deployments: a NoSQL database, and a virtual IP Multimedia Subsystem developed implementing a virtual network function. Experimental results show that our ADS can achieve a high detection and diagnosis performance.

Keywords:
Anomaly detection, system monitoring, machine learning, fault injection, SLA, diagnosis, virtualization.

1. Introduction

The development of virtualization technologies has contributed to the wide adoption of the cloud computing paradigm in various application areas. Cloud computing enables the delivery of configurable computing resources with convenient, on-demand network access to these resources. Three main categories of resources are generally provided: infrastructures, development platforms, and applications. The associated service types are respectively referred to as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

To specify the expected quality of service requirements relevant for a particular service, Service Level Agreements (SLAs) are drawn up between cloud providers and users. Such SLAs may encompass various aspects such as performance requirements and dependability properties. A *violation* of such SLAs may have severe consequences on the users and lead to potential financial penalties.

Ensuring SLAs is a challenging task for cloud providers. Although they always seek to limit the risk of SLA violation occurrence by oversizing VM allocated resources, cloud providers cannot prevent anomalies caused by unintended events such as application bugs, error propagation or flooding attacks. Since anomalies may be triggered by a large range of unexpected events, anomaly detection and diagnosis is necessary to help providers to react before failures.

Also, recovery countermeasures enabling for instance the reconfiguration, restart, or migration of the services diagnosed to be potentially at the origin of a future SLA violation need to be planned to mitigate the effects of the detected errors in order to avoid performance and dependability degradations impacting the end user experience.

In this paper, we are more particularly interested in the development of an anomaly detection system (ADS) that can efficiently address three main challenges faced by cloud providers.

First, cloud providers have to manage a large set of various types of services that are usually run with different types of workloads and execution profiles. These services may face different types of errors at runtime, and may be subject to several reconfigurations and dynamic workload changes during operations. Therefore, it would be time consuming for cloud providers to gather detailed knowledge of each individual service to detect when the service exhibits an anomalous behavior. Consequently, the anomaly detection and diagnosis mechanisms should be not only *automated* but also should not require too much effort to setup and operate.

The second challenge is related to the need to ensure that the ADS could be easily applicable to various types of services without arduous configurations. Accordingly, it should be designed to be *generic* and *service-agnostic* and should not depend on any prior knowledge of the service specification or on the types of anomalies that might affect the service.

Finally, the ADS needs to be run *online* in order to dynamically adapt to environmental and workload changes and also to enable quick decision making to identify appropriate countermeasures to be taken as soon as an anomaly is detected.

Our goal is to design and implement an ADS satisfying the aforementioned major criteria and associated challenges. Indeed, two detection objectives are targeted: i) detection of errors and preliminary symptoms that might potentially lead to SLA violations, and ii) detection of SLA violations. The detection of errors and preliminary symptoms provides early alarms of a future SLA violation. In the case that such errors are not detected, the detection of SLA violations is still relevant in order to trigger appropriate last-minute countermeasures. Besides the detection of anomalies, the goal of the proposed ADS is also to provide a high-level diagnosis of the origin of the detected anomalies (in particular, the error type and the virtual machine at the origin of the anomaly).

Different types of data can be used as inputs to perform anomaly detection and diagnosis. The ADS proposed in this paper relies on system monitoring data corresponding to software counters (such as CPU consumption, free memory...), collected through the hypervisors or the operating systems hosting the virtual machines on which the cloud services are deployed. As shown in previous works Nguyen et al. (2013); Gong et al. (2010a); Dean et al. (2012); Guan and Fu (2013); Silvestre et al. (2014); Sauvanaud et al. (2015b), such data are well suited to reflect the behavior of the monitored target system at runtime. In our context, relying on such data is also motivated by the requirement to develop generic and service-agnostic detection mechanisms that can be easily applied to different types of services. A significant result of this paper is that we analyze in different scenarios the detection and diagnosis performance when applied on data collected at the hypervisor level. We demonstrate that for some anomalies analyzing hypervisor data only, allows the detection with high accuracy. This result offers the opportunity to cloud providers to perform anomaly detection and dignosis for the running VMs while being non intrusive with respect the users (no monitoring agents are run within the VMs).

Anomaly detection is often based on machine learning (ML) algorithms Denning (1987); Lee et al. (1999); Heberlein (1995); Zhang et al. (2008); Aleskerov et al. (1997); Lee and Xiang (2001) that are well suited to fulfill the requirement to perform detection online and automatically. Both supervised and unsupervised ML algorithms can be used. With supervised learning, anomalous behaviors of the service are classified based on prior knowledge of data corresponding to normal behavior of a service and anomalies. This knowledge is acquired during an experimental execution of the service called *training phase*. Unsupervised learning does not require such knowledge and enables the detection of unknown anomalies. While our ADS can work with both algorithms, as shown in our previous work Sauvanaud et al. (2015a), in this paper we focus on supervised algorithms. Since a cloud service is mostly exhibiting a normal behavior during runtime, we developed fault injection tools for two goals: i) to inject anomalies during the training phase of machine learning models and collect data of both normal behaviors and anomalies, ii) to assess the anomaly detection and diagnosis efficiency of our ADS during a validation phase in presence of anomalies.

The key contributions of this paper include the following:

- Definition of a new ADS for cloud services enabling the detection of two types of anomalies (errors and SLA violations (SLAV)) while providing two diagnosis levels to the cloud provider (i.e., identifying the anomalous VM and the type of error causing the anomaly):

- Deployment and validation of our ADS on a VMware based cloud computing platform, with detailed sensitivity analyses illustrating its detection performance using supervised machine learning algorithms. The validation is based on two case studies coming from different industrial domains: the MongoDB database mon (2016)

and the IP multimedia subsystem (IMS) developed as a Virtual Network Function (VNF). The experimental results include a comparative analysis of the detection performance obtained with OS related monitoring data and hypervisor monitoring data.

As discussed in the related work section of this paper, anomaly detection has been the subject of an active research effort in various application domains, including in cloud services and infrastructures. One major focus of the work presented in this paper is to provide detailed insights about the implementation and the practical performance of the proposed ADS through its application to two different case studies and the execution of several sensitivity analyses. The results show a high performance of our ADS for detecting errors and SLA violations affecting cloud services, and also for diagnosing the anomalous VMs and associated types of errors. The best detection is obtained with OS related monitoring data. To the best of our knowledge, published work have a more limited scope regarding the anomaly detection and diagnosis objectives and the results presented. The specific properties of the proposed ADS make it generic and suitable to cloud infrastructures as it does not require specific knowledge of running applications neither of the functional dependencies among VM components to detect and diagnose errors.

In the following, we first present our ADS in section 2. A prototype describing an implementation of the proposed ADS is presented in section 3. Then we describe our case studies in section 4 as well as the experimentations run in order to collect validation-purposed datasets. Sections 5 and 6 provide validation results of our ADS respectively on the MongoDB and the Clearwater case studies. Sections 7 and 8 resepectively discuss related work and some limitations of our ADS. Finally, section 9 concludes and provides some directions for future work.

2. Anomaly detection system

Our ADS is associated with a target system running a service and serving users requests. It collects monitoring data from the monitored target system and processes this information to detect possible anomalies.

A cloud service is deployed on one or several VMs hosted on a virtualized infrastructure composed of several hypervisors. Each VM runs a specific application that is essential to the service execution.

In the experimental environment we use for our ADS validation, a workload generator emulates actual users requests to solicit the service. This workload generator can be used to compute and log the number of failed requests. This information is used to evaluate the service level and to train our machine learning algorithms to infer the SLAV from monitoring data.

Usually, a cloud provider does not have access to the specification of the service deployed on the infrastructure. Consequently, our ADS is not configured to have any a priori knowledge about relevant features that constitute an anomalous behavior of the target system. It actually needs to learn from system monitoring data what represents an anomalous behavior.

Our ADS is organized in three entities shown in figure 1: the monitoring entity, the data processing entity, and the fault injection entity. We first present the three entities before presenting in section 2.4 the performance measures evaluated for our ADS validation. Specifically, section 2.1 describes the monitoring entity that collects the monitoring data to be analyzed by the data processing entity. The latter is described in section 2.2. Section 2.3 describes the fault injection entity that aims at emulating errors into a service to observe the service under normal and abnormal conditions, and to gather datasets of monitoring data representing them. The datasets are used for two complementary purposes: i) to train anomaly detection and diagnosis models, and ii) to validate the detection performance of our ADS.

2.1. Monitoring

As for any computing system, the performance of a cloud service can be observed and analyzed based on several means such as system or application logs, system monitoring data, or audit trails as shown in Simache et al. (2002); Simache and Kaâniche (2001); Tan et al. (2012); Dean et al. (2012).

System monitoring data are numerical data that have the advantage not to need extra processing like pattern matching in the case of text data, in order to be handled by detection algorithms. The gathering of such data has the advantage not to depend on the implementation of the service, therefore we use monitoring data in our ADS.

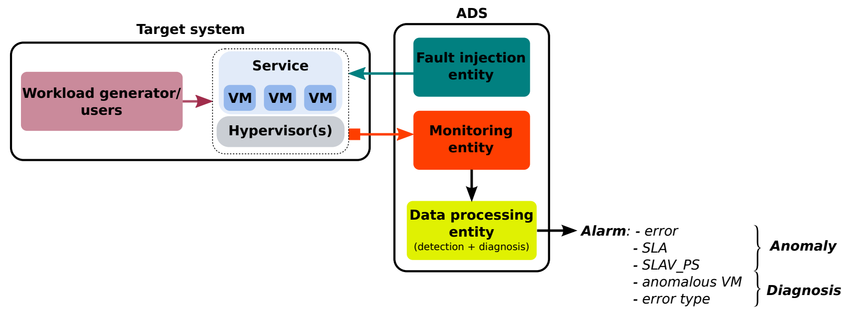


Figure 1: Target system and ADS entities.

In more details, system monitoring provides units of information about a system that are called *counters*. The actual counter values are called *metrics*. A vector of metrics collected at a given timestamp corresponds to an *observation* (also referred to as *monitoring data*).

An observation collected on the monitored system in the presence of anomaly is referred to as an *anomalous observation*, otherwise it is a *normal observation*.

We assume that anomalies result in changes in a service behavior, and these changes lead to significant variations of some system monitoring metrics, allowing their detection.

In our ADS, observations are collected periodically by means of two possible *monitoring sources*: either from the hypervisor, or from the OS of each of the VMs providing the service.

In practice, in a real deployment, only one source is to be used. Indeed, one of our objectives is to compare the detection performance of our ADS, using these two sources of data illustrated in figure 2 and described below.

- **Hypervisor monitoring**. The hypervisors hosting a service VMs can provide monitoring data related to each VM such as its memory consumption, its CPU usage or its network bandwidth. Indeed, the hypervisors grant these resources to the VMs and usually propose an Application Programming Interface (API) to fetch such data. Therefore, the *hypervisor monitoring source* does not need any tool to be installed in the VMs to collect data.

- **OS monitoring**. The OS of a service VM can also provide monitoring data if additional monitoring agents are installed in the VM. The number of available counters from the OS is more important than in the case of the hypervisor monitoring source. Indeed, the counters collected by the agents are related to different aspects of OS performance such as system buffers size and use, and in terms of memory pages state for instance. These low level VMs counters are not known by the underlying hypervisor.

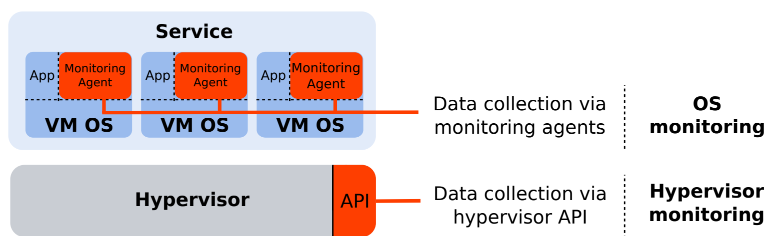


Figure 2: Sources for the collection of monitoring data from VMs hosting a cloud service. The monitoring data are presented in orange.

VM observations are analyzed by the data processing entity separately for each VM (using either OS or hypervisor monitoring data). Figure 3 illustrates this separate processing. In the figure, data are transmitted directly for data processing, and are stored in a database when needed (see section 2.2).

When an analysis is performed based on data related to a given VM, say VM_A, VM_A is referred to as the *observed VM*.

Since the VMs on which a service is deployed may implement different applications with different behaviors, processing the collected metrics per-VM enables us to individually analyze each VM without generalizing their behaviors. Also, it provides valuable information for the analysis of anomaly propagation and the analysis of the root cause of an anomaly. Anomalies may indeed propagate into several VMs interconnected through a network or close to each other in a datacenter. The propagation may be caused for instance by the reception of anomalous packets by several VMs from the same anomalous router, or by a high temperature in a particular area of a datacenter. For instance, the root cause of an anomaly could be the VM that first exhibited an anomalous behavior.

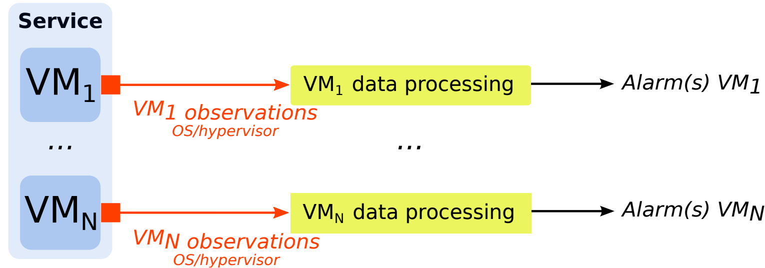


Figure 3: Processing of VM metrics.

2.2. *Data processing per VM*

This entity performs two parallel processing tasks in order to respectively detect the two types of anomalies considered in this paper: errors, and SLAVs. We first define the error and SLAV measure, before presenting the data processing tasks, followed by the detection and diagnosis models handled by each of these tasks.

2.2.1. *Definition of measures*

As per the definition stated in Avizienis et al. (2004), an *error* occurs when the system deviates from the correct service state. It is regarded in this work as the part of the system state that may lead to an SLAV. The cause of an error is a *fault*.

SLAs generally define agreements between one provider and its clients that express high level requirements related to service availability or performances ETSI (2012). Cloud SLAs overlap with SLAs of a wide range of other usage-based markets (e.g. Telecommunications) that were used for a long time beforehand. As a result, the requirements can be expressed in terms of commonly known low level metrics such as the service response time or the service throughput, or by global metrics such as the percentage of successful completion of submitted requests, in other terms, the percentage of successful requests (PSR).

In this work, we evaluate more particularly the percentage of unsuccessful requests ($PUR = 1 - PSR$) as a measure of unavailability. Thus, we consider that the SLA of a service is satisfied as long as the PUR does not overpass a maximal threshold PUR_max. An *SLA violation* (*SLAV*) occurs when PUR_max is overpassed (i.e., $PUR > PUR_max$). The PUR can be computed for a single user (i.e., user PUR) or for the set of users of a service (i.e., service PUR). In this paper, we evaluate the service PUR.

For the sake of clarity, we also hereby define a *preliminary symptom of an SLAV* (SLAV_PS) as the state describing the system behavior during a period $[t - \delta_t, t]$ where t is the time of occurrence of the SLAV. δ_t should be set according to one's detection purposes and to the nature of the SLAV (maybe the SLAV is one that can be predicted a long time ahead, in that case, δ_t should be set to 20 min for instance).

A typical evolution of the PUR before an SLAV occurrence is given in figure 4, illustrating a system experiencing a normal behavior, preliminary symptoms of SLAV (i.e., SLAV_PS) and SLAV. In this paper, we evaluate the service PUR. SLAV_PS has been investigated in our previous work Sauvanaud et al. (2016).

It can be noticed that the PUR threshold overpassing can be either caused by an anomalous network driver for instance or an enormous growth of user requests. Both cases are considered as anomalies ; in the last case scenario, the system should have been able to scale by means of load prediction systems and automatic system scaling.

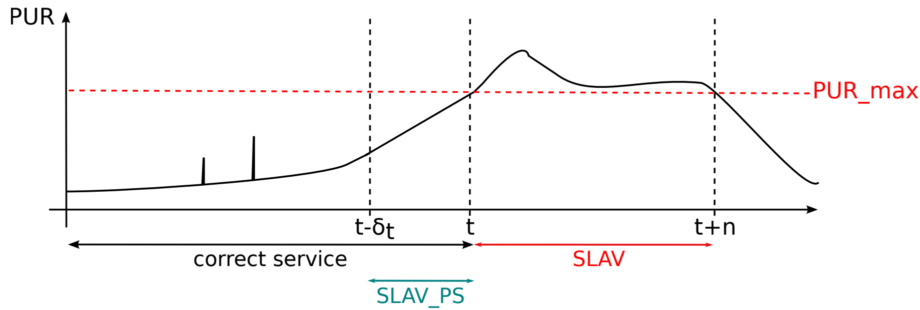


Figure 4: Typical evolution of the PUR.

2.2.2. Data processing tasks

Each parallel task classifies whether the system is experiencing a normal behavior or respectively one of the two types of anomalies (a task classifies between normal behavior and error, the other one classifies between normal behavior and SLAV). Thereupon, each task has to perform a *binary classification*.

Anomaly diagnosis requires *Multiclass classification*. in this context, a task performs detection while identifying several classes of anomaly like ”anomaly in VM_x CPU” or ”anomaly in VM_x memory”.

Two levels of diagnosis are thus possible in our ADS allowing to either diagnose the anomalous VM, or the type of error within the observed VM.

It is worth to mention that, in our approach, for each VM, we use only data collected from this VM to characterize its behavior. A question could be raised about whether data collected from a given VM can also allow to diagnose an anomaly occurring in another VM contributing to the same service. This question will be investigated during the experimental validation of the ADS in section 6.

2.2.3. Data processing models

Machine learning is a popular field of computer science aimed at implementing automatic computing procedures to learn a task without being explicitly programmed. It turned out to be extremely relevant for classification problems on numerical data Michie et al. (1994). Machine learning can be applied so as to classify behaviors corresponding to the two types of anomalies by creating *classification models* operating on system monitoring data.

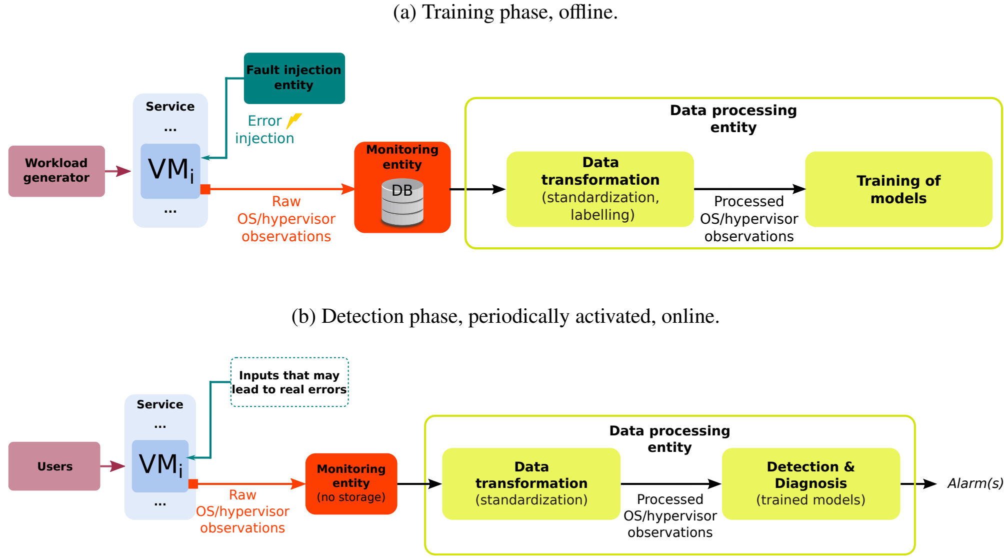
There is a large number of machine learning algorithms. The classifiers can be divided into three types depending on their learning approach: supervised, unsupervised, or semi-supervised. A supervised learning requires the provision of samples of labeled data to build classification models. Unsupervised learning relies on density or distance thresholds for instance, to create groups of data but they do not rely on labeled data. A semi-supervised learning relies on both labeled and unlabeled data samples to build classification models that still can be updated with labeled samples after their training phase.

In this paper, we concentrate on supervised learning algorithms which are widely used classifiers, also known to provide good detection performance in previous work Van Hulse et al. (2007); Farshchi et al. (2015).

Supervised learning algorithms consist of two phases, presented in figures 5 (a) and (b).

The first phase is the *training phase*, during which classification models are created *offline*. The creation of models requires samples of labeled monitoring data indicating whether or not there was an anomaly during the collection of the sample. Samples are called *training data*. They are collected by the monitoring entity and stored in a DataBase (DB in figure 5 (a)). One can note that by using labeling, data are not analyzed as time series and labeled data can be considered individually without time ordering. The more samples are provided during this phase, the more accurate is the model. One should make a sensitivity analysis to identify the minimal size of the training dataset that still allows a good detection performance. Such samples are obtained from a *training-purposed offline execution* of a target system. During such an execution, the target system is monitored while experiencing normal behaviors as well as anomalies obtained through fault injection campaigns (they are described in 2.3).

Once a model is trained, it is used during the second phase referred to as the *detection phase*.



(a) Training phase, offline.

(b) Detection phase, periodically activated, online.

Figure 5: Real deployment phases of supervised anomaly detection.

For a real deployment, the second phase corresponds to the operational phase of the system. Following offline model training phase, the model is used to detect anomalies occurring in the system itself. Observations of each VM are directly routed by the monitoring entity to the data processing corresponding to the VM and processed online.

In an experimental environment, the trained model is used to detect anomalies resulting from faults injected to validate the detection performance of the ADS (validation-purposed execution). The observations gathered during such executions are called *validation data*.

In both cases, the ADS performs *predictions* of whether a new monitoring observation belongs to a particular class of behaviors that it learned to classify. Models can be configured to predict the probabilities of class membership of an observation. We call them *prediction probabilities*. The *output* of a model is therefore one or several probabilities, depending on whether it performs a binary or multiclass classification. For instance, a model output for an observation can be a probability of 0.2 for it to be in the class labeled 1 and corresponding to the detection of an error. Given the resulting probabilities, it is then easy to set a *detection threshold* defining the limit above which an observation corresponds to a particular class. A prediction is accurate when a model predicts that an observation corresponds to an anomaly (resp. normal behavior) and some postponed analysis confirms that is was indeed an anomaly (resp. normal behavior). Such an analysis corresponds to the validation of the detection. It notably can be performed by means of i) the configuration of a workload generator and ii) the injection campaign configuration.

One can note that an offline execution can be used both for training and validation purposes. Also, we recall from section 2.2.2 that the diagnosis of an anomaly is handled by means of multiclass detection models. Thus, the diagnosis of an anomaly is also performed during the detection phase.

Finally, the current implementation of our ADS focuses on the notion of detection threshold to validate the predictions of detection models. In the end, this leads to consider binary predictions: yes or no to the detection of one type of anomaly. However, a confidence level like the one presented in Carrozza et al. (2008) could be computed from algorithms detection probabilities. This would actually be necessary to validate the location of an anomaly pointed by several VMs of a large scale system.

2.3. *Fault injection*

Fault injection is used in our study for two goals: i) the collection of service monitoring data representing both anomalies and normal behaviors in order to train detection models, ii) the validation of our ADS in the presence of anomalies. As the current definition of our ADS is based on supervised learning algorithms, the fault injection entity especially needs to cover a wide range of potential anomalies. We hereby do not propose a fault model but instead,

we use software implemented fault injection technique to emulate errors. By this means, faults are characterized according to their impact on the service resources. As one error can originate from several classes of faults, error emulation enables to enlarge the spectrum of fault coverage of our ADS.

The error emulations (sometimes also referred to as fault injections in the literature) are carried out by means of injection tools in the target service VMs. In the following we present the errors that our injection tools emulate and describe the orchestration of several injections into *injection campaigns*.

2.3.1. Error emulation

Software implemented fault injection tools are used to inject widespread abrupt anomalous behaviors that are recorded in common computing systems. Such behaviors arise when several classes of software or hardware faults are activated.

Local emulation of errors is carried out through the injections performed in services VMs. Such injections are characterized by two parameters: the type of error they emulate and their intensity level.

Five error types are emulated and distinguished according to the service resource they impact: i) high CPU consumption, ii) misuse of memory (i.e., increase of memory consumption), iii) abnormal number of disk accesses (i.e., increase of disk I/O access and synchronizations), iv) network packet loss, and v) network latency increase. They are respectively referred to as *CPU, memory, disk, network packet loss, and network latency* errors, and are described below.

CPU errors. Abnormal CPU consumptions may arise from programs encountering impossible termination conditions leading to infinite loops, busy waits or deadlocks of competing actions, which are common issues in multiprocessing and distributed systems.

Memory errors. Abnormal memory usages are common and happen when allocated chunks of memory are not freed after their use. Accumulations of unfreed memory may lead to memory shortage and system failures.

Disk errors. A high number of disk accesses, or an increase of disk accesses over a short period of time, emulate disks whose accesses often fail and lead to an increase in disk access retries. It may also result from a program stuck in an infinite loop of data writing.

Network packet loss and latency errors. Such errors may arise from network interfaces of the service or from the network interconnection of the virtualized infrastructure hosting the service. We emulate packet losses and latency increases. Packet losses may arise from undersized buffers, wrong routing policies or even firewall misconfigurations. Latency anomalies may originate from queuing or processing delays of packets on gateways or at the service level.

These four types represent all four resource domains exploited by computing systems in general and our two case studies in particular (databases and Network Function Virtualization, or NFV, architectures). They are also commonly referenced in the general literature about fault injection Kanoun and Spainhower (2008) as well as in more recent work on NFV architecture for instance Cotroneo et al. (2017).

Injection intensities correspond to a gradation of the impact magnitude of an injection in a service. Such notion is tackled in this work because preliminary experimentations showed that when not enough examples of varying amount of resource occupation were included in the training dataset, the machine learning algorithms were not efficient always at detecting and locating the injections being performed.

The calibration of injection intensities depends on the target system. Indeed, even if the VMs of a service are configured with the same VM resource template, the applications installed in each VM do not use resources in a similar way. For example, a memory injection can have a large impact on the behavior of a memory-oriented application but no impact on the behavior of a different type of application, or the same application but with a different configuration. As a consequence, high intensity injections in a VM could lead to a high value of the PUR (Percentage of Undelivered Requests) whereas the same injection in another VM could lead to a low PUR. It is therefore important to calibrate intensity levels in order for the injections to be intensive enough for the ADS to detect them, while not being too intensive for the VMs not to freeze or reboot.

Table 1 presents the intensity levels that we calibrated for our Clearwater case study presented in section 4.2.

Regarding the memory, disk and CPU injections, the intensity values of errors are constrained by the capacity of the VMs OSs. In other words, level 10 of injection (resp. level 5) is the maximum resource consumption (resp. 50% of resource) allowed by the OS before killing the execution of the injection agent.

Considering the remaining types of injections, level 10 of injection (resp. level 5) value is set so as to lead to around 99% (resp. around 50%) of PUR when applied in at least one VM.

Error type	Unit	Intensity level									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
CPU	%	10	20	30	40	50	60	70	80	90	100
Memory	%	70	73	76	79	82	85	88	91	94	97
Disk	#process	5	10	15	20	25	30	35	40	45	50
Network packet loss	%	0.004	0.008	0.012	0.016	0.02	0.024	0.028	0.032	0.036	0.04
Network latency	ms.	2	8	14	20	26	32	38	44	50	56

Table 1: Injection intensity levels.

2.3.2. Injection campaign

An injection campaign corresponds to the execution of a customizable main script that periodically performs injections in target service VMs.

An injection is defined by i) the targeted VM, ii) its error type, iii) its intensity level and iv) its duration. During a campaign, two consecutive injections are separated by a stabilization time. With regard to these parameters, a campaign consists of injecting all combinations of injections (i.e., the injection of each error type of each intensity level, in each VM).

The parameters of an injection campaign are as follows: target VMs listed in *l_vm*, error types listed in *l_type*, intensity levels, listed in *l_intensity*, an injection duration set in *inject_duration*, a stabilization time set in *pause*.

In more details, each error type of each intensity level is first injected in a first VM, then in a second VM, etc. Our campaigns execution is explained in algorithm 1. Also, figure 6 depicts an injection campaign with the following parameters: $l_vm = \{VM_1, VM_2\}$, $l_type = \{CPU\}$, and $l_intensity = \{4, 7\}$. An injection campaign ends when all error types of all intensity levels were injected in each VM.

Algorithm 1 Injection campaign

Input: l_vm, l_type, $l_intensity$, $inject_duration$, $pause$

```
for vm in l_vm do
    for err in l_type do
        for intens in l_intensity do
            injection = Injection(err, intens, inject_duration)
            inject_in_vm(vm, injection)
            sleep(pause)
        end for
    end for
end for
```

The injection duration should be long enough in order to collect a sufficient number of anomalous observations, while being short enough in order for the injection duration to be realistic. Moreover, injection campaigns with very short injection durations do not necessarily lead to SLAVs. As a consequence, while monitoring a service and running such campaigns, it is not possible to collect numerous observations of SLAV, that are however needed for the training phase of supervised algorithms. Therefore, the injection durations must be carefully calibrated so as to be long enough.

Regarding stabilization times between injections, they enable the VMs to reach a stable normal behavior after each injection. This time should be calibrated with regard to the case study behaviors. A short stabilization time could lead to an overlap of anomalous behaviors related to consecutive injected errors; which makes it difficult to diagnosis the anomalies.

2.4. Validation of the ADS detection performance

Since classification models and ADS are never perfect, we need to validate our ADS by deploying its entities as well as a target system comprising a workload generator. The ADS monitors the service, injects errors, and stores

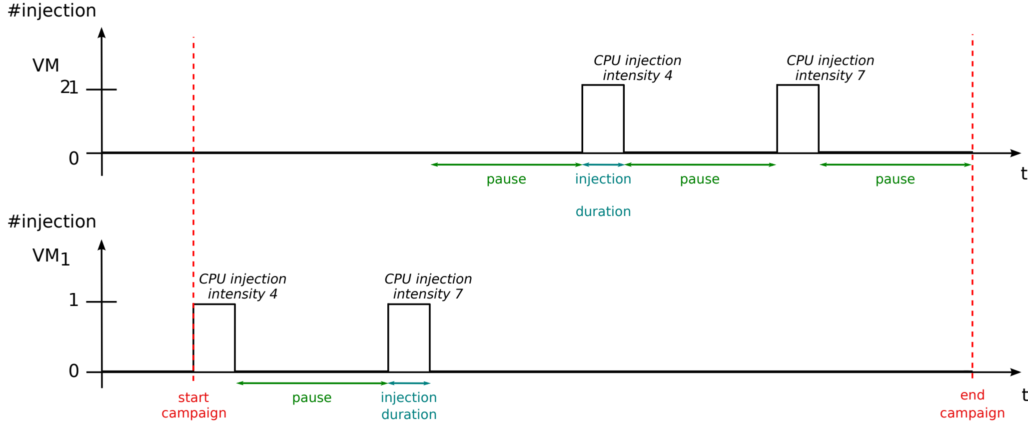


Figure 6: Injection campaign example.

normal and anomalous observations describing the service in database for further analysis. We call *experimentation* the validation-purposed execution of our ADS during a fault injection campaign on the target system. The resulting datasets are then available for validation *analyses*.

The analysis results (i.e., the ADS detection performance) are given in terms of the classical ROC (Receiver Operating Characteristic) and PR (Precision-Recall) curves. ROC and PR are widely used in anomaly detection analyses but usually separately like in Dean et al. (2012); Guan et al. (2012b); Miyazawa et al. (2015). In this work, we use both ROC and PR curves because they provide, by definition, complementary overviews.

The ROC curves present the True Positive Rate ($TPR = \frac{TP}{TP+FN}$[1]) and the False Positive Rate ($FPR = \frac{FP}{FP+TN}$) for different detection thresholds. Both rates are independent from the proportion of anomalies in the dataset being studied compared to the proportion of normal behaviors, i.e., they are independent from the dataset labels *distribution*. The results obtained by the analysis of such curves can therefore potentially be generalized to datasets with different distributions Davis and Goadrich (2006). A perfect classifier always has a $TPR = 1$, and $FPR = 0$.

The PR curves present the precision ($precision = \frac{TP}{TP+FP}$) and the recall (or TPR) for different detection thresholds. By definition of the precision, the PR curves depend on the dataset distribution. A perfect classifier always has a precision and a recall of 1.

The area under the ROC (resp. PR) curve well summarizes the ROC (resp. PR) values for all the detection thresholds Bradley (1997). A perfect classifier would have an area under the curve (AUC) of 1.

While computing the AUC obtained for an analysis (it corresponds to the run of 100 tests), we consider that the results, and thus the detection performance is: *excellent* when both the PR and ROC AUCs are above 0.90, *acceptable* when both the PR and ROC AUCs are above 0.70, and *not acceptable* when at least the PR or ROC AUCs is below 0.70.

3. Implementation

We deployed on a virtualized platform a prototype implementing the three entities of our ADS in three modules: monitoring, detection, and fault injection modules.

The platform is composed of a cloud cluster including two hypervisors and several VMs. The number of VMs is dictated by the target service needs (7 for the MongoDB case study and 3 for the Clearwater case study). In addition, the monitoring entity and the data processing entity are respectively deployed on one VM each. The monitoring entity encompasses a database centralizing data collected from the target service VMs. The data processing entity runs the processing tasks of each service VM in parallel. The workload and the fault injection entity are deployed on the same

[1]TP: True Positive, FP: False Positive, TN: True Negative, FN: False Negative

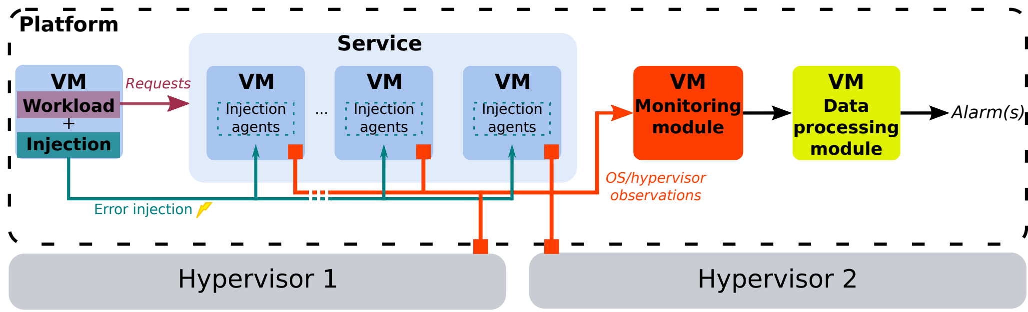


Figure 7: Virtualized platform.

VM. This facilitates the start of an experimentation which includes the launch of a workload, of the monitoring, and of an injection campaign.

The platform is a VMware vSphere 5.1 private cloud composed of 2 servers Dell Inc. PowerEdge R620 with Intel Xeon CPU E5-2660 2.20 GHz and 64 GB memory. Each server has a VMFS storage. Each VM deployed for the service implementation has 2 CPUs, a 10 GB memory, a 10 GB disk. VMs are connected through a 100 Mbps network.

The deployment of the ADS modules on the VMs is illustrated in figure 7. The modules are described in the following.

3.1. Monitoring module

For comparison purpose, monitoring data are collected from two monitoring sources (the hypervisor and the OS source) as described in section 2.1. For both monitoring sources, observations of each VM are collected every 15 sec.

The hypervisor monitoring source relies on the VMware infrastructure. Metrics are gathered from the hypervisors by means of the library Pysphere[2] that communicates with the VMware SDK. For each VM, we gather the 152 counters listed online[3].

The OS monitoring source is carried out with Ganglia monitoring agents Massie et al. (2003) installed in each service VM. Ganglia is a system specially developed for the monitoring of grid computing and has low overhead. We configured the agents so as to gather from each VM 224 counters listed online[4].

Appendix A and Appendix B provide the reader with an overview of the monitoring counters collected respectively for the hypervisor and the OS monitoring sources.

3.2. Detection module

The detection module hosts our two processing tasks (associated with the two monitoring sources) for each service VM. These tasks can be run on any machine isolated from the service VMs so as to not disturb the execution of the target system. We tested four algorithms, included in the open source Python library for machine learning Scikit-learn[5], to implement the processing tasks.

- Random Forests (configured with 200 trees),
- Neural network (configured with an hyperbolic tangent function, as in LeCun et al. (1991)), a softmax function, and a learning rate of 0.001),
- Nearest Neighbors (configured with $k = 3$),
- Naive Bayes (default configuration of the sklearn library).

Two other algorithms that are not used in this paper, SVM and Gradient Boosting, have been studied in our previous work Silvestre et al. (2015b).

[2]https://pypi.python.org/pypi/pysphere
[3]https://homepages.laas.fr/ csauvana/datasets/pysphere_vm_counters.txt
[4]https://homepages.laas.fr/ csauvana/datasets/ganglia_vm_counters.txt
[5]http://scikit-learn.org/stable/

3.3. Fault injection module

An injection campaign corresponds to the execution of a customizable main script that periodically performs injections in target service VMs. Injections are carried out by injection agents installed in these VMs. There is one injection agent for each error type in each VM of a service.

In the following, we describe the implementation of our injection agents, and the choice of the injections durations. The calibration of the intensity levels has been presented in section 2.3 with the fault injection entity.

3.3.1. Injection agents

Agents are run and stopped through an SSH connection orchestrated by the campaign main script. They emulate errors presented in section 2.3 by means of a software implementation.

CPU and disk errors are emulated using the stress test tool Stress-ng[6]. CPU injections run 2 processes (there are 2 cores in each VM) running all the stress methods listed in the tool documentation. The percentage of loading is set according to the intensity level of the injection.

Disk injections start several workers writing 50 Mo and 50 workers continuously calling the *sync* command, with an *ionice level* of 0. The number of writing workers is set according to the intensity level of the injection.

Memory injections are run by means of a python script reserving memory space while continuously checking whether the amount of memory space reserved by the script corresponds to the amount set by the intensity level of the injection.

Finally, we use the Linux kernel tools *iptables* and *tc* for the injection of network latencies on the POSROUTING chain, and *iptables* on the INPUT chain for the injection of packet losses. All network protocols are targeted.

3.3.2. Injection durations

The injection duration is calibrated so as to affect several instances of workload executions (an execution lasts less than 1sec). It is worth mentioning that a 10 min injection is long enough to observe the effect of the injection, compared to the 15 sec of monitoring period.

3.3.3. Pausing time

The stabilization time was manually calibrated during prior experimentations on our case studies. In our campaigns, it corresponds to a large time that allows us to record relatively stable monitoring metrics. Data records show that this time is short as it is close to 10 min for both our case studies. However, in the interest of collecting large datasets with enough representation of normal behavior, we set this time to be between 100 min and 30 min (for datasets collected to make minor evaluations of detection performance).

3.4. Validation method

At the end of an experimentation run for validation, the gathered observations are labeled and put in a *dataset* from which we can separately analyze the counters from the hypervisor monitoring, and the counters from the OS monitoring. The resulting datasets are respectively called the *hypervisor monitoring dataset*, and the *OS monitoring dataset* of the experimentation.

An analysis corresponds to the computation of metrics characterizing the ADS performance obtained from several runs of the training and detection phase. It can be carried out using the dataset coming from either one or two different experiments:

- When using a dataset from one experiment, we split it into a training dataset and a validation dataset by means of a random subsampling (we use 40% of observations as the training dataset and 60% for the validation dataset).
- When using two datasets from two experiments, one dataset is used as for training the models and the other one for validation .

[6] `http://kernel.ubuntu.com/~cking/stress-ng/`

An analysis can address for example the error detection performance, or the sensitivity analysis of the detection performance to the injection durations using one particular dataset. For a given analysis, the selected dataset(s) is(are) used with different subsampling and models are used with different random seeds for the initialization of their training (when needed by the implementation of the supervised algorithm).

In this paper, a *test* corresponds to a run of a training phase and a detection phase using one combination of subsampled datasets and initialized models. An analysis corresponds to 100 tests.

4. Case studies

Two case studies were selected to validate our work. One case study was selected to be representative of a well established and popular service and the second one was selected as a more recent case study which is up-to-date with brand new cloud related research topics. The first case study is MongoDB and it is representative of nowadays storage services used either for personal or business uses. Nowadays, MongoDB is notably the most popular document store solution[7]. The second case study is an example of recent telecommunication functions that make use of the cloud marketing model while Also, it is worth mentioning that both case studies are open source.

MongoDB is used to derive preliminary conclusions on the detection performance of our ADS. More detailed analyses are carried out for the Clearwater case study. For both of them we describe their deployment on our platform, the workload generator used in the experimentations, and the set of experimentations carried out for this case study together with the purpose of these experiments.

For space limitation concerns, the paper does not present SLAV related results with the MongoDB case. The SLAV related results are provided for the Clearwater case study because VNF are more SLA bounded due to their very nature related to the telecommunication domain.

4.1. MongoDB

MongoDB is a NoSQL database that provides scalable and fault-tolerant data storage. It offers 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 is run. In addition, MongoDB uses partition replication, which is the most common mechanism to enforce data durability and availability in storage systems. 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 MongoDB deployment takes into account the first scheme. In addition, our deployment relies on load balancing between replicas to limit the impact of cloud anomalies on a cluster.

In more detail, we deployed MongoDB release 2.4.8 considering two partitions ($L = 2$), and a replication factor of 2 (i.e. each partition is composed of $K + 1 = 3$ VMs). We consequently deployed a cluster of seven VMs, a query router and six document stores, ”replica1” to ”replica6”. The query router of MongoDB distributes evenly the data items between 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 service.

4.1.1. Workload for MongoDB

A workload generator for NoSQL databases loads a cluster with data and runs the workload for validation purposes. This generator allows definition of workload settings, such as document size distribution, the rate of query per second, and the distribution popularity of documents. Four types of queries can be generated, namely Create, Read, Update, and Delete (CRUD) queries.

We investigate the performance of MongoDB using the Yahoo! Cloud Serving Benchmark (YCSB) Cooper et al. (2010), a workload generator and benchmark tool for the performance of different ”key-value” and ”cloud” serving

[7] https://db-engines.com/en/ranking

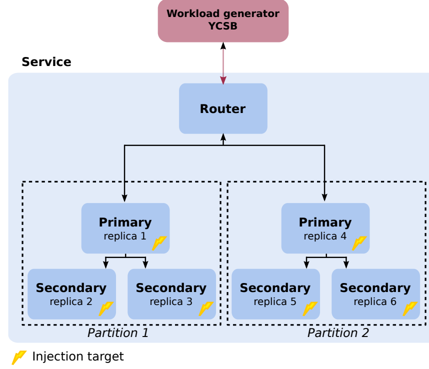


Figure 8: MongoDB deployment.

stores. 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 read/update workload. The popularity of documents follows a Zipf-like distribution. We evaluated the MongoDB performance in serving two different workloads respectively with: a constant average throughput of 3000 queries per second (called W_A), and an average throughput varying every 10 min, the number of queries per second randomly selected in the interval [500 : 3000] (called W_B).

4.1.2. MongoDB experimentations

For MongoDB, we only consider the detection of errors and the diagnosis of the type of errors affecting the observed VM. The analyses related to SLAV are carried out for the Clearwater case study. Table 2 summarizes the experiments performed for MongoDB and discussed in section 5. We first compare the error detection performance of the four learning algorithms used in the ADS (Random Forests, Neural Networks, Nearest Neighbors, Naive Bayes) to select the most suitable one to be used in our ADS to perform the rest of the analyses, related to the diagnosis of error type and the impact of a varying workload compared to a constant one.

Table 2: Description and implementation of analyses on MongoDB.

Analysis	Implementation
Error detection	The selected dataset(s) are labelled 1 if they are identified as corresponding to an error, 0 otherwise.
Algorithms comparison	The selected dataset(s) are labelled for error detection. Then, all algorithms are trained with the same training-purposed observations, and they are validated with the same validation-purposed observations.
Diagnosis of the error type	We first select a type of anomaly to be detected and dataset(s). The selected dataset(s) observations are labelled 0 if they are normal observations, and from 1 to 5 if they correspond to the selected type of anomaly in the observed VM, originating from one of our five error types.
Error detection with a varying workload	We compare the detection performance of our ADS obtained from a dataset collected while our service is serving a constant workload and obtained from a dataset collected while our service is serving a varying workload.

Table 3 shows the parameters of the fault injection campaigns carried out in the two experimentations that we will present for MongoDB including all combinations of injections. The analyses use OS monitoring data collected from these experimentations.

4.2. Clearwater

4.2.1. Description

The service is an open source VNF named Clearwater. Clearwater is an open source implementation of an IMS for cloud platforms. It provides voice and video calls based on the Session Initiation Protocol (SIP), and messaging applications. It implements key standardized interfaces and functions of an IMS (except a core network) which enable

Table 3: Injection campaign parameters for the two experimentations.

Experiment	Campaign parameters
$\mathcal{M}$ (15 days of monitoring, constant workload)	• $l_vm = \{replica1, replica2, replica3, replica4, replica5, replica6\}$ • $l_type = \{CPU, memory, disk, latency, packet_loss\}$ • $pause$ = 100 min • $l_intensity = \{1 : 10\}$ • $inject_duration$ = 10 min • Workload W_A
$\mathcal{N}$ (15 days of monitoring, varying workload)	• $l_vm = \{replica1, replica2, replica3, replica4, replica5, replica6\}$ • $l_type = \{CPU, memory, disk, latency, packet_loss\}$ • $pause$ = 100 min • $l_intensity = \{1 : 10\}$ • $inject_duration$ = 10 min • Workload W_B

industries to easily deploy, integrate and scale an IMS. Clearwater encompasses six software components, namely Bono, Sprout, Homestead, Homer, Ralf, and Ellis shown in figure 9.

Bono is the SIP proxy implementing the Proxy-Call/Session Control Functions (P-CSCF). It handles the users requests and routes them to Sprout. It also performs Network Address Translation traversal mechanisms.

Sprout is the IMS SIP router receiving requests from Bono and routing them to the adequate endpoints. It implements some Serving-CSCF (S-CSCF) and Interrogating-CSCF (I-CSCF) functions and gets the required users profiles and authentication data from Homestead. Sprout can also call application servers and actually contains itself a multimedia telephony (MMTel) application server, whose data are stored in Homer (when calls are configured to use its services).

Homestead is an HTTP RESTful server. It either stores Home Subscriber Server (HSS) data in a Cassandra database and masters the data (i.e. information about subscribed services and locations), or pulls data from another IMS compliant HSS. The HSS mirror function is considered as part of the I-CSCF and S-CSCF functions.

Thus, Bono, Sprout, and Homestead work together to control the sessions initiated by users and handle the entire CSCF.

Homer is an XML Document Management Server (XDMS) server with an XML Configuration Access Protocol Server (XCAP) interface, and runs a Cassandra database. It stores configuration information about the MMTel services.

Ralf is the Charging Trigger Function (CTF). It bills the events collected by Bono and Sprout and reports them to a Charging Data Function server (this server is not included in the Clearwater project).

Ellis is a provisioning portal offering a web interface to users for testing purposes.

Clearwater scales-out horizontally by means of a simple domain name system (DNS) load balancing mechanism. Our testbed encompasses Bono, Sprout, Homestead and Homer, each of which is deployed on one VM (see figure 9). In this study, the billing function is not configured, so Ralf is not included in our testbed, neither is the testing component Ellis. We focus our work on Bono, Sprout and Homestead constituting the entire CSCF: we perform injections and provide evaluation results for these three VMs.

4.2.2. Workload for Clearwater

IMS workloads can be emulated by means of the SIPp benchmark[8]. The benchmark contains a workload that can be configured with a number of calls per second (i.e. a load parameter) to be sent to the IMS and a scenario. The execution of a *scenario* corresponds to a call. A scenario is described in terms of SIP transactions in XML. A SIP transaction corresponds to a SIP message to be sent and an expected SIP response message. A call fails when a transaction fails. A transaction may fail for two reasons: either a message is not received within a fixed time window (i.e., the timeout), or an unexpected message is received. Unexpected messages are identified by the HTTP error codes

[8]http://sipp.sourceforge.net/index.html

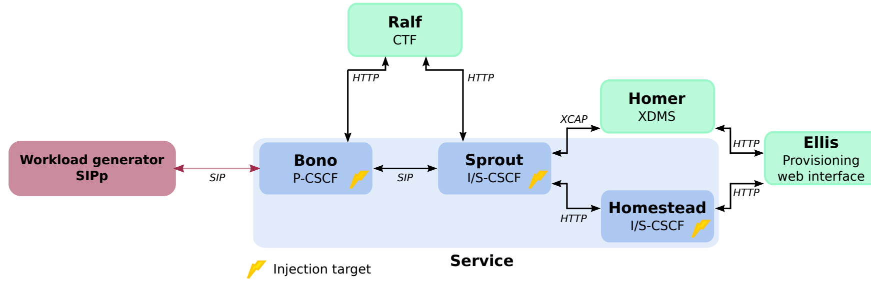


Figure 9: Clearwater deployment.

500 (Internal Server Error), 503 (Service Unavailable) and 403 (Forbidden). In this case study, the PUR corresponds to the percentage of unsuccessful calls.

The scenario run for our experimentations simulates a standard call between two users and encompasses the standard SIP REGISTER, INVITE, UPDATE, and BYE messages. The scenario is available online[9]. Timeouts are set to 10 sec as in Cao et al. (2015).

4.2.3. Clearwater experimentations

For Clearwater, we will present in section 6 a more complete set of experimentations encompassing the analyses of the two types of anomalies (errors and SLAVs), and the two diagnosis levels (identification of the anomalous VM and the type of error within the observed VM). In addition, we run several sensitivity analyses to investigate the impact of various parameters on the detection performance of our ADS. Moreover, we compare the detection performance when using hypervisor monitoring data or OSs monitoring data. Finally, we evaluate the lifetime of the ADS trained models. In other words, we evaluate the time after which the detection performance of the ADS decreases and the models need to be trained again. Such analyses are summarized in table 4.

In order to carry out these analyses, we ran four experiments from which we collected four datasets, referred to as $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$. The configuration of these experiments is described in table 5. In the first experiment, all combinations of injections are considered. The resulting dataset $\mathcal{A}$ is used as a reference for comparison. The second experiment is run to analyse the lifetime of the detection models one month after the training phase, i.e., if their performance is still acceptable. Consequently, $\mathcal{B}$ is collected one month after $\mathcal{A}$. The third experiment is run to evaluate the impact of the injection durations on the detection performance. Consequently, 4 min injection durations are considered in $\mathcal{C}$ compared to 10 min for dataset $\mathcal{A}$. The fourth experiment is run to evaluate the lifetime of models after one week of detection. The same fault injection durations are considered as for $\mathcal{B}$. Indeed, they are longer than in $\mathcal{C}$ and enable to collect more anomalous data. To this aim, we collected the corresponding dataset $\mathcal{D}$ one week after $\mathcal{B}$. The campaigns parameters of our experimentations are presented in table 3.

5. MongoDB validation results

Validation of our ADS using MongoDB is first performed with the four popular supervised machine learning algorithms (listed previously) with respect to their error detection performance and their execution times. The most suitable algorithm (Random Forests algorithm) is then used to assess our ADS error detection and diagnosis performance.

When not mentioned otherwise, the datasets collected from the experiments are randomly subsampled for each analysis carried out, so that anomalous observations for each anomaly type represent 2% of the total amount of observations.

As similar conclusions are obtained for the various replicas, we show the results of only one replica. These results was validated by means of 10-folds cross validation.

For space limitation concerns, we do not present SLA-related results for the case of MongoDB. The reader can refer to our previous work Silvestre et al. (2015b) for further detail about this point.

[9] https://homepages.laas.fr/csauvana/sipp_scenario/issre2016_sipp_scenario.xml

Table 4: Description and implementation of analyses on Clearwater datasets.

Analysis	Implementation
Error detection	The selected observations are labelled 1 if they are identified as corresponding to an error, 0 otherwise.
SLAV detection	The selected observations are labelled 0 if they are normal, and 1 if they correspond to an SLAV.
Diagnosis of the anomalous VM	We first select a type of anomaly to be detected and dataset(s). The selected observations are labelled 0 if they are normal, and from 1 to 3 if they correspond to the selected type of anomaly originating from one of three observed VM (i.e., Bono, Sprout and Homestead).
Diagnosis of the error type	We first select a type of anomaly to be detected and dataset(s). The selected dataset(s) observations are labelled 0 if they are normal observations, and from 1 to 5 if they correspond to the selected type of anomaly in the observed VM, originating from one of our five error types.
Sensitivity to the intensity level	The detection models are trained on normal observations and on anomalous observations representing injection intensity levels 4 or 7. The validation of the models is however performed on a dataset with normal observations and anomalous observations representing all seven intensity levels.
Sensitivity to the datasets sizes	Several subsamplings of the same dataset are performed to obtain data subsets containing the same distribution of anomalous observations but with a varying total number of observations. The detection performances associated to each data subset (split into training and validation subsets) are then evaluated.
Sensitivity to the injection duration	The detection performances obtained from datasets related to experimentations with different injection durations are compared.
Lifetime of detection models	Models are trained from a dataset $\mathcal{X}$ collected during a first experimentation. They are then validated using a dataset $\mathcal{Y}$ collected during an experimentation n days after the end of the first experimentation. The detection performance obtained from this study is compared to the detection performance obtained from the study of $\mathcal{X}$ randomly split and used for both training and validation.
Comparison of the monitoring sources	The detection performance obtained from the study of the hypervisor monitoring data of an experimentation is compared to the detection performance obtained from the study of a OS monitoring data of the same experimentation.

Table 5: Injection campaign parameters for the four experimentations.

Experiment	Campaign parameters
All experiments	• $l_vm = \{Bono, Sprout, Homestead\}$ • $l_type = \{CPU, memory, disk, latency, packet_loss\}$
$\mathcal{A}$ (10 days of monitoring)	• $l_intensity = \{1 : 10\}$ • $inject_duration = 10$ min • $pause = 100$ min
$\mathcal{B}$ (2 days of monitoring)	• $l_intensity = \{5, 10\}$ • $inject_duration = 10$ min • $\mathcal{B}$ was recorded 1 month after the end of $\mathcal{A}$ • $pause = 40$ min
$\mathcal{C}$ (2 days of monitoring)	• $l_intensity = \{5, 10\}$ • $inject_duration = 4$ min • The injection campaign was run twice in order for $\mathcal{C}$ to gather approximatly as many observations as $\mathcal{B}$ • $pause = 35$ min
$\mathcal{D}$ (2 days of monitoring)	• $l_intensity = \{5, 10\}$ • $inject_duration = 10$ min • $\mathcal{D}$ was recorded 1 week after the end of $\mathcal{B}$ • $pause = 30$ min

5.1. Comparative analysis of supervised learning algorithms

We assessed and compared four different types of popular supervised machine-learning algorithms to analyze their error detection efficiency, namely Random Forests, Neural Networks, Nearest Neighbors and Naive Bayes. The results

reported in figures 10 present the ROC and PR AUCs computed using dataset $\mathcal{M}$ (whose characteristics are presented in Table 3, using in particular a constant workload) for an error detection without diagnosis of the anomalous VM nor the error type. In our context, the best results are obtained with Random Forests and Nearest Neighbors with an average ROC and PR AUC above 0.95. The two other algorithms exhibit a low error detection performance (e.g., the average PR AUC is around 0.35 for Neural Networks).

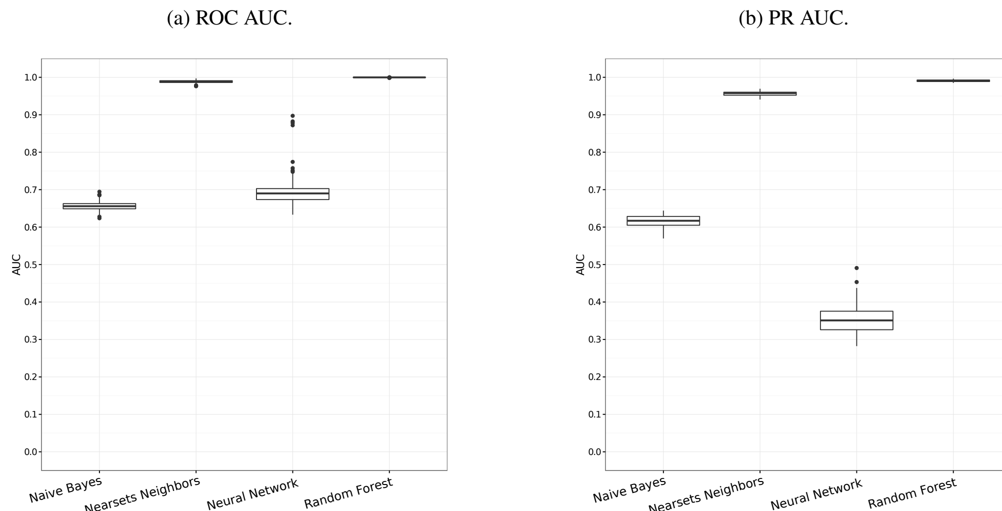


Figure 10: Comparison of binary detection results using four different supervised learning algorithms.

To provide better insights about the training and detection times of these algorithms, Table 6 presents the mean times for the processing of a single observation (composed of 227 counters), computed during the training phase of the algorithms and the detection phase. These times are computed on a 64 bits Intel Core i7 2.10 GHz processor. The computed mean values correspond to a sample of 100 executions of training and detection.

It can be noticed that the training time of Neural Networks is 10 times higher than for the other algorithms. The associated detection time remains low with a mean of 0.168 ms. The Naive Bayes is the algorithm with the lowest training and detection times with only 0.010 ms and 0.015 ms. The longest detection time (around 0.82 ms) is observed with the Nearest Neighbors algorithm. The algorithm indeed computes the Euclidean distances between the query vector (i.e. the new observation on which to make a prediction) and all points from the training data. In comparison, the Random forests algorithm for instance, only needs to evaluate some inequalities once the trees are grown, which is only few CPU instructions. This time is still acceptable in our context, especially considering that in our experimentations the system monitoring period is set to 15 sec. Finally, the Random Forests has a training time mean of 0.919 ms and a detection time almost 10 times lower. The latter is higher compared to the Naive Bayes algorithm. However, it remains low and acceptable in our context.

Considering the criteria related both to the error detection performance and to the training and detection execution times, we can conclude that Random Forests is the best algorithm to be used online, in particular due to its low detection time and high error detection efficiency.

5.2. Diagnosis of the error type: datasets $\mathcal{M}$ and $\mathcal{N}$

Figures 11 and figures 12 present the ROC and PR AUCs for error detection with a diagnosis of the error type occurring in the observed VM respectively for the datasets $\mathcal{M}$ and $\mathcal{N}$ (whose characteristics are presented in Table 3, using in particular a constant and a varying workload) based on the Random Forests algorithm.

The error detection performance with a diagnosis of the error type is excellent for both datasets $\mathcal{M}$ and $\mathcal{N}$ corresponding to observations collected respectively under a constant and a varying workload. Average ROC and PR AUCs are indeed above 0.99. The varying workload executed during the collection of $\mathcal{N}$ hardly affects the detection and diagnosis performance as all the results remain higher than 0.95 for both ROC and PR AUCs.

Phase	Measures	Naive Bayes	Neural Network	Random Forests	Nearest Neighbors
Training	Mean	0.010	10.322	0.919	0.064
	Standard deviation	0.004	1.824	0.130	0.017
Detection	Mean	0.015	0.168	0.102	0.82
	Standard deviation	0.005	0.045	0.021	0.193

Table 6: Training and detection times (ms).

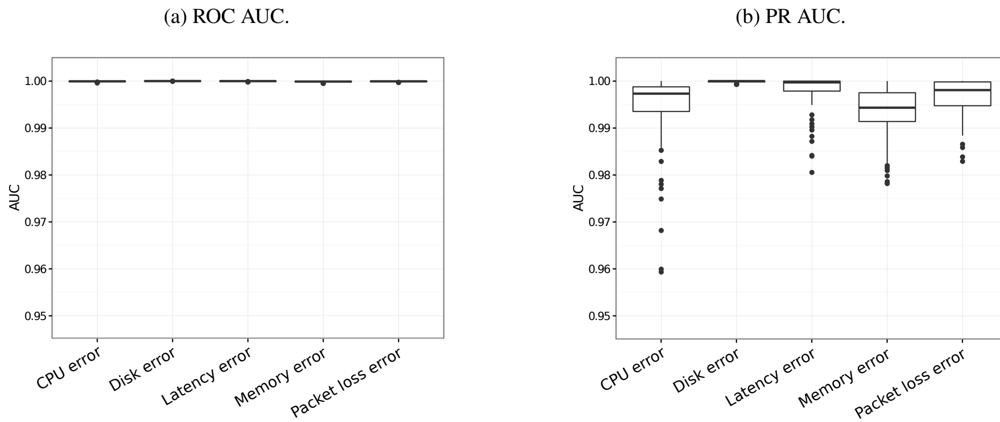


Figure 11: Error detection with a diagnosis of the error type occurring in the observed VM using dataset $\mathcal{M}$

.

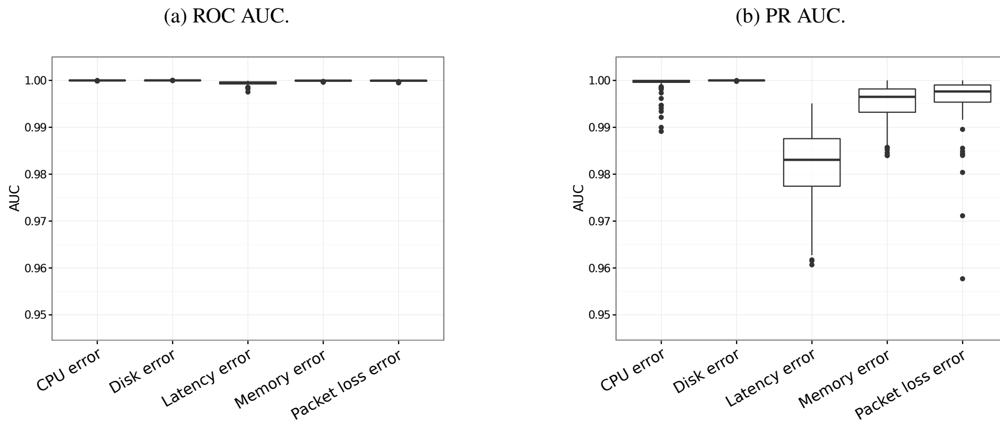


Figure 12: Error detection with a diagnosis of the error type occurring in the observed VM using dataset $\mathcal{N}$

.

5.3. Discussion

With this case study we evaluated the error detection performance of our ADS using several common machine learning classifiers and observations collected based on an OS monitoring. The main conclusions derived are the following:

- In terms of detection performance and associated training and detection times, the Random Forests algorithm is the most adapted to our context. We validated the adequacy of the Random Forests algorithm for detecting the injected errors and diagnosing the error type occurring in the observed VM, under both a constant and varying workload.
- The results reported in this section are based on the observations collected from only one replica. Similar results are obtained when using the observations from the other replicas. This is potentially due to the fact that all replicas run a similar application in our experimental setup and the workload between all replicas is well balanced.
- Furthermore, we analyzed the top 10 counters that have the most significant impact on the error detection and diagnosis performance of the ADS[10]. This analysis showed that these counters depend on the error type to be detected and diagnosed. Also, we observed that a wide range a memory counters is among the top 10 lists. We believe that this is due to the fact that the MongoDB database is memory oriented. Indeed, as all the data are mapped in memory, MongoDB enables the memory counters to be good predictors of anomalous behaviors in the database.

A further validation of our ADS using the Random Forests algorithm is presented in the next section for the Clearwater case study.

6. Clearwater validation results

This section presents the validation results related to the Clearwater case study. Section 6.1 focuses on error detection, while section 6.2 presents the results related to the detection of SLA violations. A comparative analysis with OS monitoring data is also presented.

Following a similar method as described in section 5.1, the Random Forests algorithm turns out to be the algorithm leading to the best detection performance. Results are not presented due to space limitation. It was recorded however that they are slightly different from the results in section 5.1 in that the Nearest Neighbors algorithm provides very poor results. Thus, the Random Forests algorithm is used for all the following ADS analyses.

When not mentioned otherwise, the results are based on hypervisor monitoring data and anomalous observations in datasets are randomly subsampled for each analysis to represent 2% of the amount of normal observations. These results was validated by means of 10-folds cross validation.

6.1. Detection of errors

This section is aimed at presenting the error detection and diagnosis performance of our ADS and includes several sensitivity analyses related e.g., to the distribution of anomalous observations in the dataset or to the fault injection duration.

6.1.1. Data distribution sensitivity: dataset $\mathcal{A}$

As discussed in section 2.4, the distribution of anomalous observations in the considered dataset is likely to influence the ADS detection performance. To illustrate this impact, figures 13 (a) and (b) present the ROC and PR AUCs computed for a binary error detection using data observed in Bono, Sprout and Homestead, with different distributions of anomalous observations in the dataset varying from 2% to 50%. The corresponding datasets are randomly subsampled from dataset $\mathcal{A}$. An increase of PR AUCs around 0.02 in average is reported for all observed VMs when the percentage of anomalous observations in the dataset increases from 2% to 50%. Nevertheless, the average detection performance remains excellent in our context for all observed VMs and all anomaly distributions (higher than 0.94). As expected, the ROC AUCs remain stable. It is noteworthy that this result is not biased by the duration of the experimentation which is long enough in our context (10 days).

Also, these results, as the following ones, show that the study of either the ROC curves alone or the precision and recall alone, does not provide a thorough insight on the detection performance of an ADS. Indeed, in this case, while both are relatively high, the PR curves show that even though there are only a few false positive alarms, not all anomalies are detected.

[10]They are selected across all trees of a model using the Gini impurity in order to measure the quality of the trees splits.

(a) ROC AUCs. (b) PR AUCs.

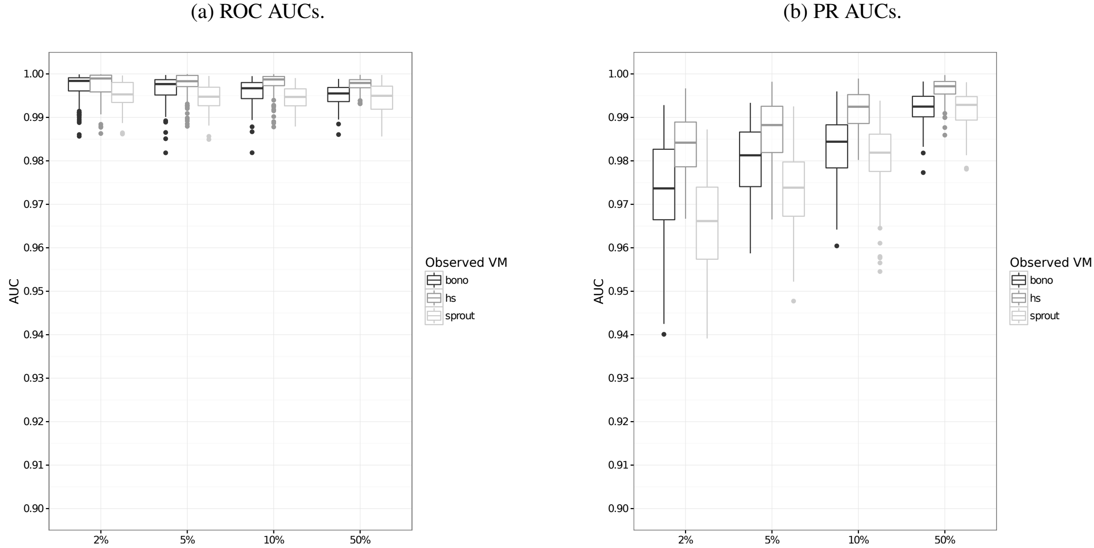


Figure 13: Binary error detection with several data distributions.

6.1.2. Diagnosis of the anomalous VM: dataset $\mathcal{A}$

Figures 14 (a) and (b) plot the ROC and PR AUCs computed for the detection of errors with the diagnosis of the anomalous VM based on the data recorded from each observed VM. In other words, for each observed VM data (represented with different color boxes) we compute the ADS performance for the detection of errors occurring in Bono, Sprout, and Homestead (shown on the x-axis).

The performance of the ADS for the detection of errors with a diagnosis of the anomalous VM is relatively high. The mean ROC AUCs are about 0.99 and the averaged PR AUCs are above 0.95 for Sprout and Homestead, and above 0.91 for Bono. We also notice that similar results are obtained with a binary detection. Indeed, the averaged ROC AUCs are as good as in figure 13 (a).

Comparing the results depending on the observed VM, it can be noticed that while the detection performance is relatively of the same order of magnitude from the three VMs, it is slightly more difficult to detect errors and diagnose the anomalous VM based on the monitoring data collected from Bono, even for errors occurring in this VM. On the other hand, the best performance is obtained with Homestead as the observed VM, even when errors are injected in the other VMs.

Thus we conclude as a response to the question asked in section 2.2.2 that using data collected from a given VM also allows to diagnose an anomaly occurring in a different VM contributing to the same service. We also notice that the VM at the end of the CSCF chain has the best knowledge and the best prediction capability about the state of the other VMs of the chain. Lower detection and diagnosis capabilities are obtained when the observed VM is located at the beginning of the chain (Bono).

6.1.3. Diagnosis of the error type: dataset $\mathcal{A}$

Figure 15 (a) and (b) present ROC and PR AUCs computed for an error detection with a diagnosis of the error type in the observed VM. The results are globally satisfactory with an average value above 0.90, with the following exceptions.

Network latency errors are detected with a lower detection performance, between 0.82 and 0.85 for both ROC and PR AUCs, in particular when the observed VMs are Sprout and Homestead. These errors are well detected from Bono, which is a proxy. This is due to the larger impact that the proxy has on the requests handling by the entire CSCF chain (Bono, Sprout and Homestead compose this chain in that order, see figure 9). An increasing latency indeed may affect the communication with Sprout. The network packet loss detection provides the worst results when it is performed based on the monitoring of Sprout with the average ROC and PR AUCs around 0.89. The reason must be linked to

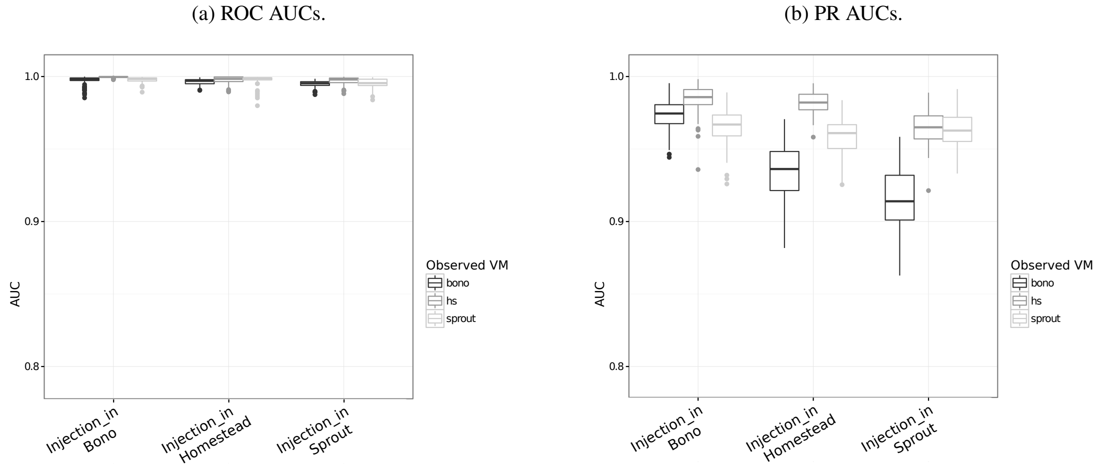


Figure 14: Error detection with a diagnosis of the anomalous VM.

some implementation details of the routing function hosted in Sprout that masks the impact of the error on monitoring data.

We can conclude that according to the code executed on each VM, errors are not detected with the same detection performance. We also notice a lower detection performance in the case of a detection with a diagnosis of the error type in the observed VM, compared to the binary detection (see figures 13). However, the diagnosis efficiency results are still relatively high and promising.

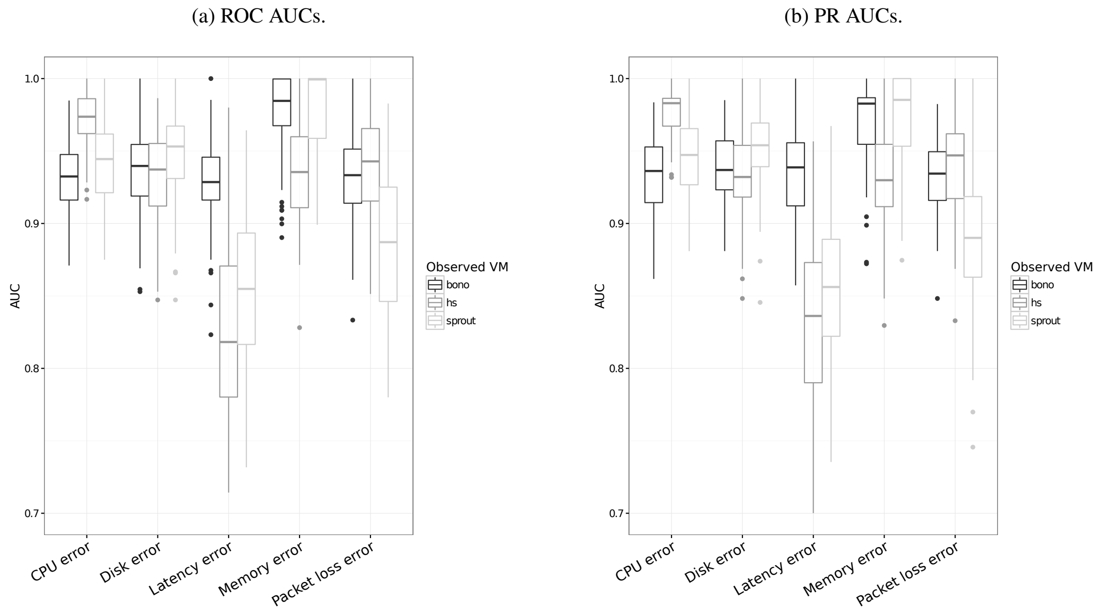


Figure 15: Error detection with a diagnosis of the error type.

6.1.4. Sensitivity to the injection intensity level: dataset $\mathcal{A}$

Figures 16 (a) and (b) present the ROC and PR AUCs computed for the error detection with a diagnosis of the error type in the observed VM for a training phase run on $\mathcal{A}$ subsampled so as to keep only anomalous observations with an error intensity level of 5 or 10 and for a detection phase on the entire dataset $\mathcal{A}$.

These figures are to be compared with figures 15 (a) and (b) where all the intensity levels of $\mathcal{A}$ are represented in the training dataset.

If we consider the ROC AUCs, the detection performances of disk and memory errors are similar to results of figure 15 (a). The detection of CPU errors is however lower but remains acceptable. Nonetheless, the detection performance of latency and packet loss errors is largely affected. They are acceptable when Bono is the observed VM, but not for Sprout and Homestead. PR AUCs show the same tendency as for the ROC AUCs, except that they are extremely low for the detection of latency and packet loss errors.

The detection performance gets lower when the training dataset does not include all injection intensity levels, especially for Sprout and Homestead which exhibit averaged ROC AUCs below 0.80. Therefore, a full injection campaign must be run for the sake of the training of our Random Forests models.

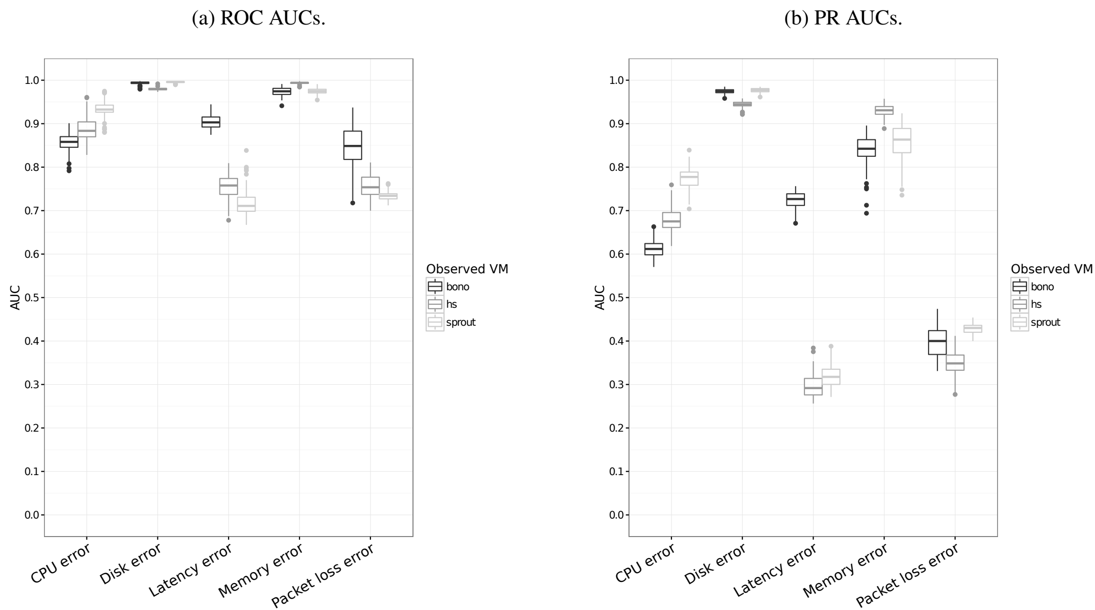


Figure 16: Error detection with a diagnosis of the anomalous VM with a training dataset comprising observations of only two intensity levels of errors represented in $\mathcal{A}$ and a detection on all $\mathcal{A}$ seven intensity levels.

6.1.5. Sensitivity to the injection duration: dataset C

To analyze the impact of fault injection duration on the ADS performance, figures 17 (a) and (b) present the ROC and PR AUCs corresponding to the detection of injected errors with a diagnosis of the error type with 4 min injection durations (i.e. dataset C is used). These figures are to be compared with figures 15 (a) and (b) that present a similar analysis while considering 10 min injection durations. We observe a lower detection performance with ROC AUCs varying between 0.97 and 0.71 across all VMs.

We performed a sensitivity analysis on the dataset size that we do not show in this work. Results show that the difference in dataset sizes for results in figures 15 and in figures 17 does not solely explain the low detection performance.

The results of a similar analysis addressing error detection with a diagnosis of the anomalous VM, with 4 min fault injection durations, are also presented in figures 18 (a) and (b). These figures are to be compared with figures 14 (a) and (b) obtained with injection durations of 10 min. Again, the detection performance is lower when shorter injection

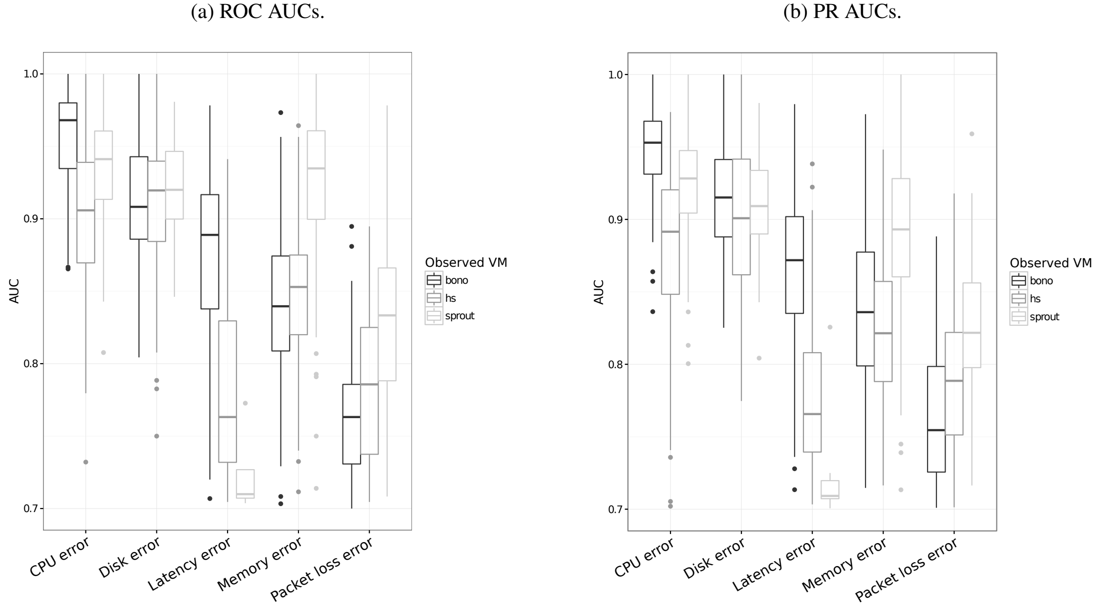


Figure 17: Error detection with a diagnosis of the error type, and a dataset with 4 min injection durations.

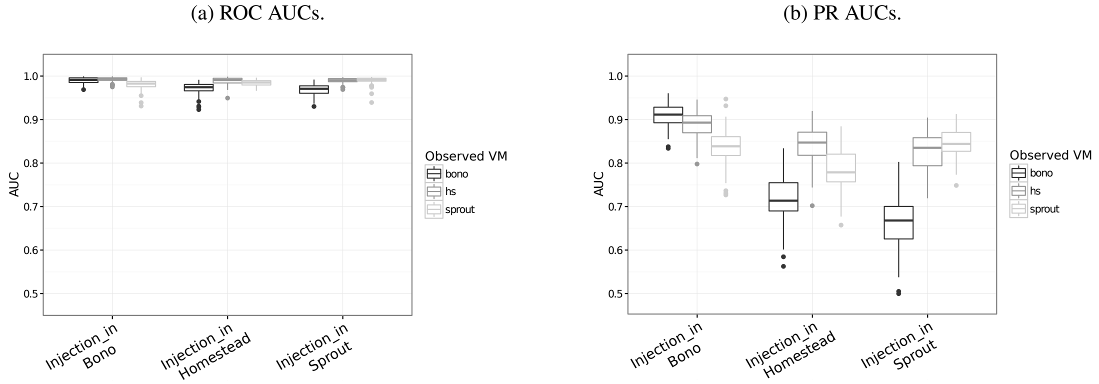


Figure 18: Error detection with a diagnosis of the anomalous VM, and a dataset with 4 min injection durations.

durations are considered. The difference regarding the average ROC AUCs for the 4 min case and the 10 min case is important and can be higher than 0.10. We conclude that the detection performance indeed depends on the injection durations.

An experimentation with 20 min injection durations was run. Results show that the detection performance is similar to those related to the 10 min durations case.

6.1.6. Lifetime of detection models: datasets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{D}$

A critical issue when using machine learning algorithms concerns the frequency at which the detection models should be updated with a new training phase to take into account more recently collected data. We carried out several sensitivity analyses to answer this question. A first analysis consisted in carrying out a binary error detection using one month old models to detect anomalies based on current observations (i.e., the models are trained on dataset $\mathcal{A}$ and the detection performance is assessed using dataset $\mathcal{B}$. An excellent detection performance was obtained with

average ROC and PR AUCs higher than 0.90 for all observed VMs. However, the results obtained in the case of a detection with a diagnosis of the error type show that the models require a new training for the detection performance to remain acceptable for all error types. More acceptable detection results are obtained with observations collected earlier, one week after the training of models (i.e., using datasets $\mathcal{B}$ and $\mathcal{D}$). The averaged ROC AUCs are excellent and the average PR AUCs are excellent for the majority of error types, except for the following cases: i) latency errors when they occur in Homestead and Sprout, and ii) memory errors in Bono.

We performed similar analyses of the lifetime of detection models with observations collected through OS monitoring. Figures 19 (a) and (b) present ROC and PR AUCs computed for the error detection with a diagnosis of the error type with a training phase based on $\mathcal{A}$ observations and a detection performed on $\mathcal{B}$. The detection performance is higher than when using hypervisor monitoring data. In most cases, the ROC and PR AUCs are excellent. The lowest average AUC values obtained for the detection of latency errors in Sprout and Homestead remain acceptable. In this case using OS monitoring data, the detection models still provide a good detection performance after one month.

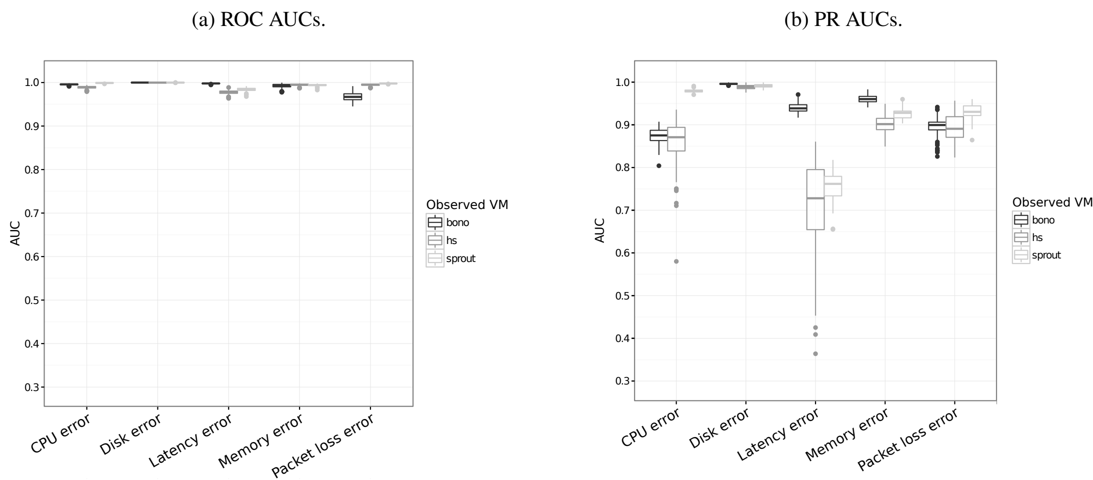


Figure 19: Error detection with a diagnosis of the error type with OS monitoring data and with a training on $\mathcal{A}$ and a detection on $\mathcal{B}$ (the observations of $\mathcal{B}$ are collected one month after the end of the data collection of $\mathcal{A}$).

6.2. *Detection of SLA violations (SLAV)*

So far, we have only analyzed the performance of our ADS for the detection and diagnosis of errors. In this section we focus on SLA violations. The SLAV detection alarms must be handled quickly because the service is already degraded when the alarm is being raised. In the following analyses, PUR_max is set to 2% for SLAV definition.

6.2.1. *Diagnosis of the anomalous VM: dataset $\mathcal{A}$*

Figures 20 present the ROC and PR AUCs for the SLAV detection with a diagnosis of the anomalous VM. These figures show that it is possible to efficiently pinpoint the anomalous VM from each observed VM. The ROC AUCs are indeed excellent for all observed VMs with a mean above 0.90. The PR AUCs are acceptable but they depend on the observed VM. We note that the best detection performance with average values of PR AUCs above 0.87 is obtained with Homestead observations.

Thus, the detection performance for the SLAV detection with a diagnosis of the anomalous VM is generally good. It is however lower compared to the error detection with diagnosis performance results (see section 6.1.2). Also, the same conclusion is obtained as in section 6.1.2 concerning the fact that the observed VM leading to the best diagnosis of the anomalous VM is Homestead.

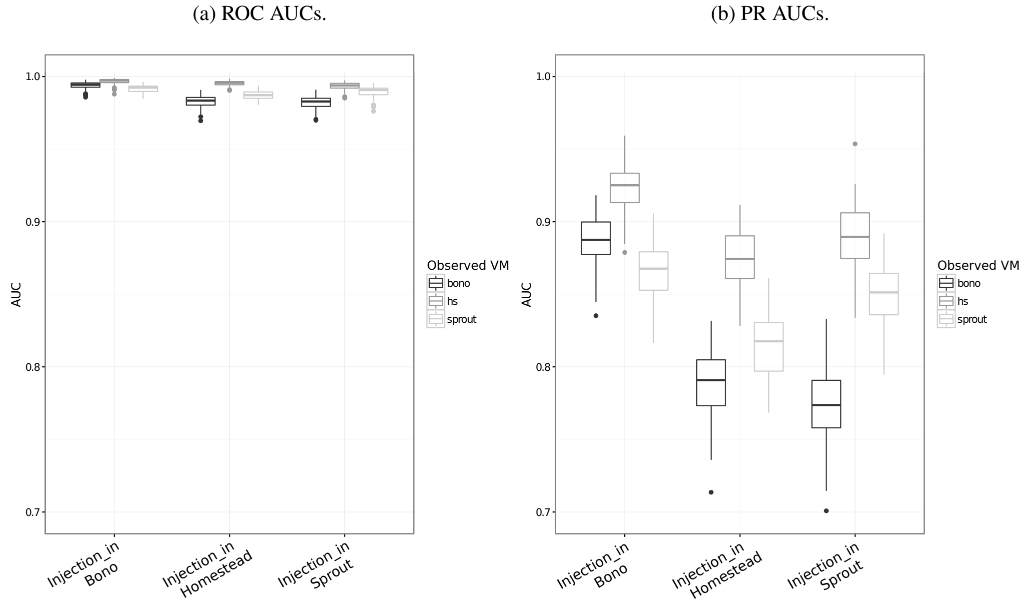


Figure 20: SLAV detection with a diagnosis of the anomalous VM.

6.2.2. Comparison of hypervisor and OS based monitoring: dataset $\mathcal{A}$

Figures 21 (a) and (b) present the ROC and PR AUCs obtained for the SLAV detection with a diagnosis of the anomalous VM, using OS monitoring data. These figures are to be compared with figures 20 (a) and (b) which present the results of SLAV detection with hypervisor monitoring data. The detection performance is higher in the case of OS monitoring data. We can conclude that the numerous OS counters enable a better description of the VM system and increase the detection performance SLA violations with a diagnosis of the anomalous VM.

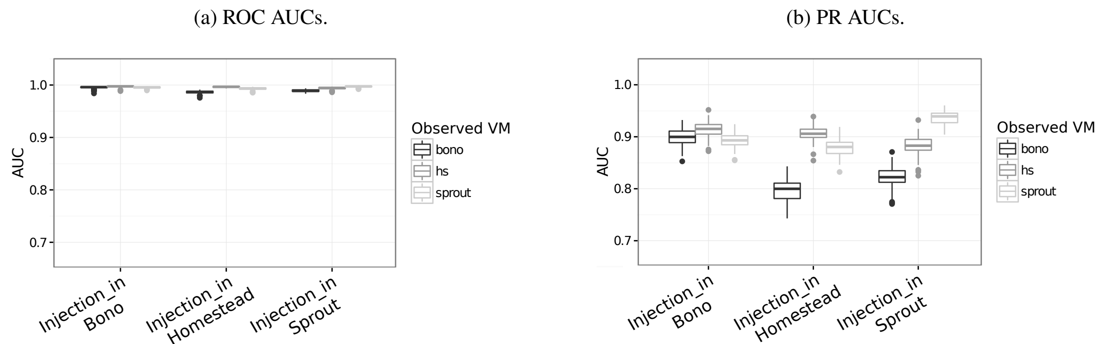


Figure 21: SLAV detection with a diagnosis of the anomalous VM using OS monitoring data.

6.2.3. Diagnosis of the error type: dataset $\mathcal{A}$

Figures 22 (a) and (b) present the ROC and PR AUCs computed for the SLAV detection with a diagnosis of the error type.

Regarding the ROC AUCs, the detection performance is excellent with values above 0.98 across all VMs. The averaged PR AUCs are also excellent with means values above 0.90. The detection performance is better than when

considering the same analysis for the error detection (see figures 15).

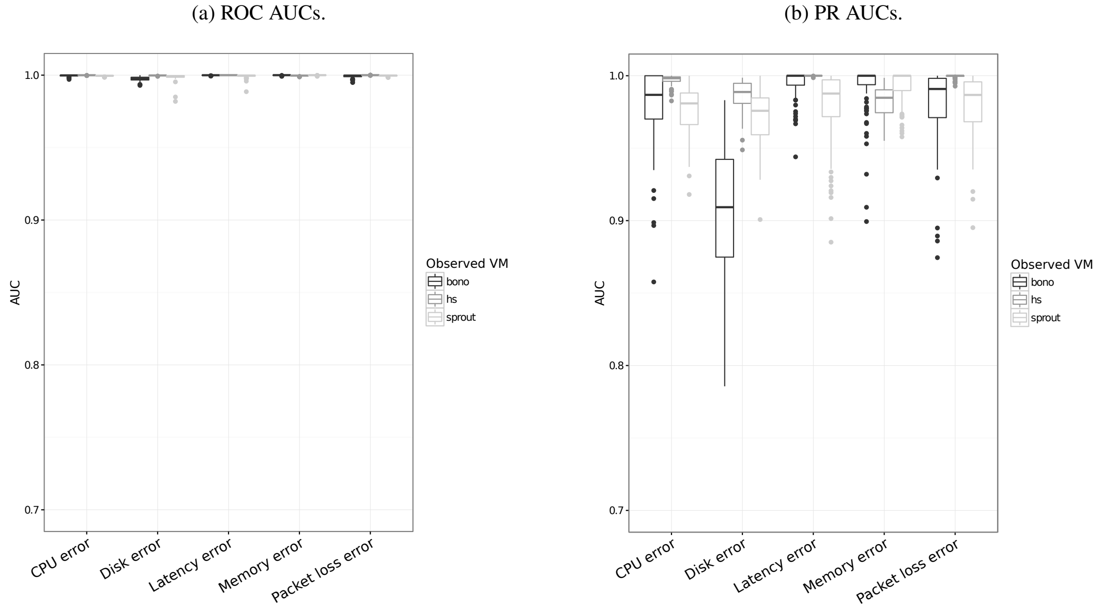


Figure 22: SLAV detection with a diagnosis of the error type.

6.2.4. Sensitivity to the injection durations: dataset C

The ROC and PR AUCs related to SLAV detection with a diagnosis of the error type, and for injection durations of 4 min (i.e. we use dataset C) are similar to the ones obtained in figure 20 (a) and (b) for injection durations of 10 min.

6.3. Discussion

We hereby further discuss the main lessons learned from the Clearwater case study:

- Results demonstrate that similar anomalies are not detected with the same detection and diagnosis performance depending on the VM where the errors are injected (Bono, Sprout or Homestead). We conclude that the application installed on a VM may impact the performance of our ADS and therefore that the ADS performance depends on the case study. In this paper, we validated our ADS on two different case studies, a NoSQL database and an IMS. Although the detection performance varies, it remains good, and even excellent in many cases.
- In more detail, we observed that it is possible to detect and diagnose from the observation of a VM (say VM A) an anomaly that occurs in an other VM (say VM B). We conclude that there exists a propagation of the effect of the anomaly occurring in VM B to VM A, and this propagation can be isolated by means of monitoring data characterizing VM A. Besides, the best anomaly detection and diagnosis performance is not always obtained based on the observations collected from the VM in which the errors were injected.
- We also noticed that, using Homestead hypervisor monitoring data leads to the best diagnosis performance for localizing the VM at the origin of an error or an SLAV. It is however difficult to conclude that Homestead monitoring data in general appear to be good inputs for the detection in general. Indeed, more data needs to be gathered on a full Clearwater deployment in order to state whether this remark is due to the role of Homestead as a Cassandra database or due to its location at the end of the CSCF chain. Notably the Ralf component, communicating with both Bono and Sprout, should be studied in order to evaluate whether it can

detect anomaly occurring in Bono and Sprout. It is noteworthy that the same conclusion was not confirmed for all cases by the results based on OS monitoring data. Additional counters in OS monitoring certainly provide enough information for the detection from Bono and Sprout to be as efficient.

- OS monitoring data turn out to provide better performance than hypervisor monitoring data. As a matter of fact, we investigated the most important counters for the Random Forests models to take classification decision. Looking at the OS monitoring counters, the most relevant ones are related to the TCP/IP protocol. However, this group of counters is not available from the hypervisor monitoring because it cannot be known directly from an underlying hypervisor. We conclude that this group of counters is important for the detection.
- Moreover, we performed similar analysis of the important counters used for the diagnosis of each type of error. It turns out that, as expected from the MongoDB case study, the important counters are different depending on the type of error at the origin of an anomaly to diagnose. Also, these counters are different from the most important counters identified for the MongoDB case study. Clearly, the most relevant counters depend on the application (memory oriented database or router for instance). This explains the difference observed in the detection and diagnosis performance between the MongoDB and Clearwater case studies.
- The lifetime of detection models is shorter when using hypervisor monitoring data. Using the OS monitoring data, models still provided a good detection performance even after one month of execution, both for the detection of errors and SLAV.

  Consequently, using a supervised machine learning algorithm such as Random Forests, the more monitoring counters are used, the better is the detection and diagnosis performance. We however note that this is not the case for unsupervised learning algorithms such as clustering. Indeed such algorithms are based on metrics like distances or densities between observations that tend to be large and similar in large dimensional spaces (also known as the curse of dimensionality). A wide range of monitoring counters should in this case be handled with analyses of several subspaces of counters.

We conclude that an ADS aimed at detecting errors and SLAV in different cloud services should be validated on several case studies.

7. Related work

The wide adoption of virtualization in several application domains gave rise to a large body of research work covering a variety of topics, dealing for instance with elastic resource contention issues of VMs or services Kundu et al. (2012); Matsunaga and Fortes (2010); Bodík et al. (2009); Silvestre et al. (2015a), server reconfiguration Cerf et al. (2016), resources and energy management Chase et al. (2001); Berral et al. (2010), and security Bhat et al. (2013); Gander et al. (2013). Other works also propose frameworks dedicated to the processing of large datasets (or *big data*) sourced from cloud infrastructures such as Pop (2016). As a result, we present in this section related works that deal with anomaly detection in large computing systems with no regard to their deployment paradigm.

Research works related to anomaly detection are mostly distinguished by the data processed for anomaly detection, and the algorithms applied on data to actually perform the detection.

Concerning the processed data, several papers make use of logs as in Salfner and Malek (2007); Watanabe et al. (2012); Xu et al. (2009); Liang et al. (2006), other make use of audit traces Denning (1987); Lee et al. (1999); Heberlein (1995); Mukherjee et al. (1994) (both logs and audit traces correspond to text files) and the third large family of data is monitoring data Dean et al. (2012); Tan et al. (2012); Zhang et al. (2013); Guan et al. (2012a); Nguyen et al. (2013) (monitoring data are numerical data).

Regarding the detection algorithms, several techniques are used based for instance on the statistics, the probability theory or information theory, such as in Wang et al. (2010); Williams et al. (2007); Lee and Xiang (2001); Gong et al. (2010b); Nguyen et al. (2013). Machine learning is a field of computer science which algorithms can solve classification problems. The following contributions respectively deal with supervised and unsupervised learning algorithms for anomaly detection.

Supervised algorithms. The comparative works in Van Hulse et al. (2007); Farshchi et al. (2015) propose evaluations of popular algorithms such as Support Vector Machines (SVM), supervised neural networks, and decision

trees for anomaly detection. Liang et al. (2007) compares three algorithms for the detection of preliminary symptoms of failures in IBM BlueGene/L, namely SVM, threshold nearest neighbors and a rule-based classifier named RIPPER. In Aleskerov et al. (1997), neural networks are used for the detection of credit card frauds. The Fa system proposed in Duan et al. (2009) centralizes failure signatures represented in terms of system resource consumption. It relies on Classification and Regression Trees (CART) and SVM to classify new behaviors into normal behaviors or failures. Cohen et al. (2004) applies Tree-Augmented Bayesian Networks (TAN) created from labeled data to detect service violations in web services. In Tan et al. (2012), the authors focus on clouds VM systems and they propose a solution to anticipate system attributes values by means of Markov models whose transitions depend on both the current state and the previous state, then the attributes are classified using TANs. In Zhang et al. (2008), the authors use Random Forests (RF) to detect network intrusions. Finally, ALERT Tan et al. (2010) is a system predicting anomalies using a clustering phase (it is an unsupervised learning algorithm) that enables to build groups of execution contexts, and a second phase where decision trees are trained to detect anomalous contexts.

Unsupervised algorithms. In Dean et al. (2012), anomaly detection is tackled for cloud infrastructures using unsupervised neural networks, making the assumption that anomaly has preliminary symptoms expressed in system monitoring data. Regression models are used in Cherkasova et al. (2009) for a workload change detection tool. The tool inspects the CPU consumption of a server as well as the number of client transactions. Principal Component Analysis is applied on application logs in Xu et al. (2009) and in Lakhina et al. (2004) for the detection of anomalies in network traffic. Clustering refers to several algorithms used to group data. Clustering is applied in the anomaly detection tool eCAD Zhang et al. (2013) using system monitoring data of clouds VMs. Likewise, Leung and Leckie (2005); Mazel (2011) use clustering algorithms for anomaly detection in network traffic. vNMF Miyazawa et al. (2015) Niwa et al. (2015) is a tool that detects anomalous behaviors in VNFs using unsupervised neural networks. Clusters are considered anomalous when they are too small and far apart from the other normal clusters.

Other related works address the **diagnosis of anomalies**. Perfcompass Dean et al. (2014) focuses on fault diagnosis with a temporal analysis of fault propagation, depending on system calls. It is dedicated to fault diagnosis once an anomaly is detected by other means. DAPA performance diagnostic framework Kang et al. (2012) models the existing relationships between application performances and underlying system metrics to detect SLA violations. The SLA violation diagnostic is performed with an unsupervised learning algorithm. PeerWatch Kang et al. (2010), an anomaly detection and diagnosis tool, uses a correlation analysis to model the relationship existing between the components of distributed applications, and detects an anomaly when the correlation between these components drops significantly. FChain Nguyen et al. (2013) monitors the execution of distributed applications to detect performance anomalies and to pinpoint the faulty component by reconstructing the propagation patterns of abnormal change points.

In our previous work Silvestre et al. (2015b) we defined an approach relying on machine learning for error detection in databases. In Sauvanaud et al. (2015b) we presented a new unsupervised algorithm based on clustering for error detection in computing systems. In our most recent work Sauvanaud et al. (2016) we evaluated an approach relying on supervised learning for the case of the cloud service Clearwater while tackling the diagnosis of an SLA violation or preliminary symptoms of SLA violations. An online diagnosis was also proposed while either pinpointing an anomalous VM or a high workload toward the service. In this paper, we largely improved this approach into an ADS for the detection of three types of anomaly and two diagnosis, whereas related work focus on the detection of only one type of anomaly or one single type of diagnosis. Moreover, the diagnosis is provided online. Our ADS also has the advantage not to depend on strict synchronisation constraints between several components and a diagnosis entity. Finally, unlike some of the presented contributions, our ADS solely relies on monitoring data. Thus, it has the advantage not to require any specification of the cloud services.

8. Limitation

The ADS proposed in this paper suffers from the intrinsic limitations of supervised learning algorithms especially proceeding from the need of a complete training dataset in order to make good predictions on previously known (and labeled) data classes.

In our context, this issue can be mitigated by means of frequent re-training of models with the monitoring data already being collected continuously for the sake of detection. Besides, the promising results obtained with models from supervised learning algorithms are good and provide extremely few false positive. In that sense, we are extremely interested in mitigating the drawback of a complete training dataset by also exploiting models from unsupervised

algorithms. In that scenario, the models from supervised learning would detect the most common anomalies. As for the outputs of the models from unsupervised learning (e.g., based on the clustering of observations like in our previous work Sauvanaud et al. (2015a)), they would be considered on periods during which the models from supervised learning would not detect any anomaly, and possibly missing some unknown ones.

In addition, similarly to our evaluation in , further validation should consider to train models from supervised learning with a subset of known error types and evaluate the detection performance of the ADS with a validation dataset containing all known error types.

Finally, while exploiting error emulation, we consider that all classes of faults located in the guest OS, or in the virtual resources are leading to the proposed four error types. Consequently, injection campaigns need to tackle all four types of errors to assess the validity of our ADS. It can be noticed, however, that the impact of application crashes or of virtual CPU pining on real CPU need to be evaluated. The error emulation presented in this paper enables to add these with a very quick development time for our future work.

9. Conclusions and Future work

In this paper, we defined and experimentally evaluated the performance of a new anomaly detection system, ADS, for cloud services. This ADS allows the detection of erroneous components and SLA violations. Two diagnosis levels are considered: at the VM level (i. e., diagnosis of the anomalous VM), and a more detailed diagnosis, giving the error type at the origin of an anomaly within the anomalous VM.

Our ADS relies on i) service-agnostic system monitoring data, ii) machine learning algorithms to classify anomalous and normal behavior and to perform diagnosis (in this paper we focused on supervised learning algorithms), iii) fault injection to collect training data including anomalous samples to train the detection and diagnosis models and to validate our ADS.

We compared the performance of our ADS based on two sources of monitoring data: data provided by the hypervisor through its API, or data collected from the OS by means of agents installed in each VM involved in the service.

Data processing is carried out in a decentralized way, in parallel for all hypervisors, or for all VMs, depending on the type of monitoring data, making the detection process very efficient, with a short detection time. In particular, we have compared the efficiency of several classification algorithms and concluded that the Random Forests algorithm provides in our context the best tradeoff in terms of detection efficiency and training and detection time.

We validated our ADS using two target systems: MongoDB database and Clearwater IMS. Several examples of sensitivity analyses are presented to analyze the characteristics of our ADS, some of them are summarized hereafter:

- The diagnosis of the error type or of the anomalous VM causing an erroneous behavior of a VM, or an SLA violation is possible.
- Our ADS can adapt to constant and varying workloads.
- The two types of anomalies can be detected with a good detection performance; this performance depends on the node from which the detection is based.
- While both hypervisor and OS monitoring data generally lead to high detection and diagnosis performance, OS monitoring data always leads to better results. This is due to the fact that OS monitoring data encompasses low level metrics notably TPC/IP metrics that are not provided in hypervisor monitoring data.
- The detection and diagnosis models need to be trained periodically to adapt to dynamic changes of the execution environment and workloads. However, using OS monitoring data seems to give good prediction during longer periods of time.

In this paper, we focused on the training on several models associated with the detection and diagnosis of anomaly for each VM of a service. As a future work, It would also be relevant to study how to combine the alarms raised by different detection models to derive comprehensive decision rules taking into account potential correlation of the alarms raised by several models. For instance, an alarm is raised by the ADS only if a majority of detection

models integrated in the ADS raise an alarm. Clustering algorithms can also be investigated based on the probabilities associated to these alarms.

Concerning the classes of errors considered, so far in our work, errors are related to the VM, we plan to inject errors related to the hypervisor as well as errors related to the applications running the service. More generally, other classes of errors should be addressed to enlarge the spectrum of errors considered so far.

Moreover, with regard to the created models for our different VMs, we plan to evaluate the detection perfomance of such models giving data related to VMs supporting the same role. The aim is to exploit one single model for all VMs providing the same redundant role among a service.

Finally, in this paper SLA violation is assessed in terms of the percentage of unsuccessful requests (i.e. a measure af availability), therefore we additionally plan to work on the assessment of different metrics of SLA such as the service latency and throughput, like we did in our previous work Silvestre et al. (2015b).

Appendix A. Overview of hypervisor monitoring counters

```
cpu_costop
cpu_demand
cpu_entitlement
cpu_idle
cpu_ready
cpu_run
cpu_swapwait
cpu_used
cpu_wait
rescpu_actav1
rescpu_actpk1
rescpu_runpk1

mem_activewrite
mem_compressed
mem_compressionRate
mem_decompressionRate
mem_llSwapInRate
mem_overheadTouched
mem_swapinRate
mem_swapoutRate
mem_zipped
mem_zipSaved

datastore_maxTotalLatency
datastore_numberReadAveraged
datastore_numberWriteAveraged
datastore_totalWriteLatency
datastore_write
disk_busResets_naa
disk_commandsAborted_naa
disk_maxTotalLatency
disk_numberReadAveraged_naa
disk_numberWrite_naa
disk_read
disk_write_naa
virtualDisk_largeSeeks_scsi
virtualDisk_mediumSeeks_scsi
virtualDisk_numberReadAveraged_scsi
virtualDisk_numberWriteAveraged_scsi
virtualDisk_readOIO_scsi
virtualDisk_read_scsi
virtualDisk_writeOIO_scsi
virtualDisk_write_scsi

net_broadcastRx_nic1
net_broadcastTx_nic1
net_bytesRx_nic1
net_droppedRx_nic1
net_droppedTx_nic1
net_multicastRx_nic2
net_packetsRx_nic1
net_packetsTx_nic1
net_received_nic1
net_transmitted_nic1
```

Appendix B. Overview of OS monitoring counters

```
cpu_idle
cpu_intr
cpu_nice
cpu_num
cpu_user
cpu_wio
load_fifteen

mem_cached
mem_free
mem_hardware_corrupted
mem_mapped
mem_writeback
swap_free
swap_total

bytes_in
bytes_out
disk_free
disk_free_absolute_rootfs
disk_free_percent_rootfs
disk_total
diskstat_sda_io_time
diskstat_sda_percent_io_time
diskstat_sda_read_bytes_per_sec
diskstat_sda_read_time
io_busymax
io_max_wait_time
io_nread
io_nwrite
io_reads
io_writes

icmp_inaddrmasks
icmp_inerrors
icmp_inmsgs
icmp_outtimeexcds
ip_forwdatagrams
ip_fragcreates
ip_fragfails
ip_fragoks
ip_inaddrerrors
ip_indiscards

pkts_in
pkts_out
rx_bytes_eth0
rx_bytes_eth1
rx_bytes_lo
rx_drops_eth1
rx_errs_eth1
rx_pkts_eth1
tcp_activeopens
tcp_attemptfails
tcp_currestab
tcp_estabresets
tcp_inerrs
tcp_insegs
tcp_outrsts
tcp_outsegs
tcp_passiveopens
tcp_retrans_percentage
tcpext_arpfilter
tcpext_delayedacklost
tcpext_embryonicrsts
tcpext_listendrops
tcpext_tcpabortonsyn
tcpext_tcpabortontimeout
tcpext_tcpdsackoforecv
tcpext_tcpmemorypressures
tcpext_tcpprequeued
tcpext_tcpsackshiftfallback
tcpext_twkilled
tx_bytes_eth0
tx_bytes_eth1
tx_bytes_lo
tx_drops_eth1
tx_errs_eth1
tx_pkts_eth1
udp_indatagrams
udp_inerrors
udp_outdatagrams
udp_sndbuferrors

vm_kswapd_skip_congestion_wait
vm_pgmajfault
vm_pgpgin
vm_pgpgout
vm_vmeff
```

References

MongoDB, 2016. URL http://www.mongodb.org/.

E. Aleskerov, B. Freisleben, and B. Rao. Cardwatch: a neural network based database mining system for credit card fraud detection. In *Computational Intelligence for Financial Engineering (CIFEr), 1997., Proceedings of the IEEE/IAFE 1997*, pages 220–226, Mar 1997. doi: 10.1109/CIFER.1997.618940.

A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

J. L. Berral, I. n. Goiri, R. Nou, F. Julià, J. Guitart, R. Gavaldà, and J. Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 215–224, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0042-1. doi: 10.1145/1791314.1791349. URL http://doi.acm.org/10.1145/1791314.1791349.

A. H. Bhat, S. Patra, and D. Jena. Machine learning approach for intrusion detection on cloud virtual machines. *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, 2(6):56–66, 2013.

P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855533.1855545.

A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

L. Cao, P. Sharma, S. Fahmy, and V. Saxena. Nfv-vital: A framework for characterizing the performance of virtual network functions. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 93–99, Nov 2015. doi: 10.1109/NFV-SDN.2015.7387412.

G. Carrozza, D. Cotroneo, and S. Russo. Software faults diagnosis in complex ots based safety critical systems. In *2008 Seventh European Dependable Computing Conference*, pages 25–34, May 2008. doi: 10.1109/EDCC-7.2008.26.

S. Cerf, M. Berekmeri, B. Robu, N. Marchand, and S. Bouchenak. Cost function based event triggered model predictive controllers-application to big data cloud services. In *55th IEEE International Conference on Decision and Control*, 2016.

J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev.*, 35(5):103–116, Oct. 2001. ISSN 0163-5980. doi: 10.1145/502059.502045. URL http://doi.acm.org/10.1145/502059.502045.

L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Automated anomaly detection and performance modeling of enterprise applications. *ACM Trans. Comput. Syst.*, 27(3):6:1–6:32, Nov. 2009. ISSN 0734-2071. doi: 10.1145/1629087.1629089. URL http://doi.acm.org/10.1145/1629087.1629089.

I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251270.

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL http://doi.acm.org/10.1145/1807128.1807152.

D. Cotroneo, L. D. Simone, and R. Natella. Nfv-bench: A dependability benchmark for network function virtualization systems. *IEEE Transactions on Network and Service Management*, PP(99):1–1, 2017. ISSN 1932-4537. doi: 10.1109/TNSM.2017.2733042.

J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

D. J. Dean, H. Nguyen, and X. Gu. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 191–200, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1520-3. doi: 10.1145/2371536.2371572. URL http://doi.acm.org/10.1145/2371536.2371572.

D. J. Dean, H. Nguyen, P. Wang, and X. Gu. Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, Philadelphia, PA, June 2014. USENIX Association. URL https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/dean.

D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, Feb 1987. ISSN 0098-5589. doi: 10.1109/TSE.1987.232894.

S. Duan, S. Babu, and K. Munagala. Fa: A system for automating failure diagnosis. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1012–1023. IEEE, 2009.

ETSI. Technical report 103 125 v1.1.1. cloud; slas for cloud services. http://www.etsi.org/deliver/etsi_tr/103100_103199/103125/01.01.01_60/tr_103125v010101p.pdf, 2012.

M. Farshchi, J. G. Schneider, I. Weber, and J. Grundy. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 24–34, Nov 2015. doi: 10.1109/ISSRE.2015.7381796.

M. Gander, M. Felderer, B. Katt, A. Tolbaru, R. Breu, and A. Moschitti. *Anomaly Detection in the Cloud: Detecting Security Incidents via Machine Learning*, pages 103–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16, Oct 2010a. doi: 10.1109/CNSM.2010.5691343.

Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16. IEEE, 2010b.

Q. Guan and S. Fu. Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 205–214, Sept 2013. doi: 10.1109/SRDS.2013.29.

Q. Guan, C.-C. Chiu, Z. Zhang, and S. Fu. Efficient and accurate anomaly identification using reduced metric space in utility clouds. In *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*, pages 207–216, June 2012a. doi: 10.1109/NAS.2012.30.

Q. Guan, Z. Zhang, and S. Fu. Ensemble of bayesian predictors and decision trees for proactive failure management in cloud computing systems. *Journal of Communications*, 7(1):52–61, 2012b.

L. Heberlein. Network security monitor (nsm)–final report. 1995. URL `http://seclab.cs.ucdavis.edu/papers/NSM-final.pdf`.

H. Kang, H. Chen, and G. Jiang. Peerwatch: A fault detection and diagnosis tool for virtualized consolidation systems. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 119–128, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0074-2. doi: 10.1145/1809049.1809070. URL `http://doi.acm.org/10.1145/1809049.1809070`.

H. Kang, X. Zhu, and J. L. Wong. Dapa: Diagnosing application performance anomalies for virtualized infrastructures. In *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Berkeley, CA, 2012. USENIX. URL `https://www.usenix.org/conference/hot-ice12/dapa-diagnosing-application-performance-anomalies-virtualized-infrastructures`.

K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008. ISBN 047023055X, 9780470230558.

S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta. Modeling virtualized applications using machine learning techniques. *SIGPLAN Not.*, 47(7):3–14, Mar. 2012. ISSN 0362-1340. doi: 10.1145/2365864.2151028. URL `http://doi.acm.org/10.1145/2365864.2151028`.

A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 219–230, New York, NY, USA, 2004. ACM. ISBN 1-58113-862-8. doi: 10.1145/1015467.1015492. URL `http://doi.acm.org/10.1145/1015467.1015492`.

Y. LeCun, I. Kanter, and S. A. Solla. Second order properties of error surfaces: Learning time and generalization. In *Advances in Neural Information Processing Systems*, pages 918–924, 1991.

W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *Security and Privacy, 2001. S P 2001. Proceedings. 2001 IEEE Symposium on*, pages 130–143, 2001. doi: 10.1109/SECPRI.2001.924294.

W. Lee, S. Stolfo, and K. Mok. A data mining framework for building intrusion detection models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 120–132, 1999. doi: 10.1109/SECPRI.1999.766909.

K. Leung and C. Leckie. Unsupervised anomaly detection in network intrusion detection using clusters. In *Proceedings of the Twenty-eighth Australasian Conference on Computer Science - Volume 38*, ACSC '05, pages 333–342, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. ISBN 1-920-68220-1. URL `http://dl.acm.org/citation.cfm?id=1082161.1082198`.

Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 425–434, June 2006. doi: 10.1109/DSN.2006.18.

Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 583–588. IEEE, 2007.

M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30:2004, 2003.

A. Matsunaga and J. A. B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 495–504, May 2010. doi: 10.1109/CCGRID.2010.98.

J. Mazel. *Unsupervised network anomaly detection*. PhD thesis, INSA de Toulouse, 2011.

D. Michie, D. J. Spiegelhalter, C. C. Taylor, and J. Campbell, editors. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, Upper Saddle River, NJ, USA, 1994. ISBN 0-13-106360-X.

M. Miyazawa, M. Hayashi, and R. Stadler. vnmf: Distributed fault detection using clustering approach for network function virtualization. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 640–645, May 2015. doi: 10.1109/INM.2015.7140349.

B. Mukherjee, L. Heberlein, and K. Levitt. Network intrusion detection. *Network, IEEE*, 8(3):26–41, May 1994. ISSN 0890-8044. doi: 10.1109/65.283931.

H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward black-box online fault localization for cloud systems. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 21–30, July 2013. doi: 10.1109/ICDCS.2013.26.

T. Niwa, M. Miyazawa, M. Hayashi, and R. Stadler. Universal fault detection for nfv using som-based clustering. In *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*, pages 315–320, Aug 2015. doi: 10.1109/APNOMS.2015.7275446.

D. Pop. Machine learning and cloud computing: Survey of distributed and saas solutions. *CoRR*, abs/1603.08767, 2016. URL `http://arxiv.org/abs/1603.08767`.

F. Salfner and M. Malek. Using hidden semi-markov models for effective online failure prediction. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 161–174, Oct 2007. doi: 10.1109/SRDS.2007.35.

C. Sauvanaud, G. Silvestre, M. Kaâniche, and K. Kanoun. Data Stream Clustering for Online Anomaly Detection in Cloud Applications. In *11th European Dependable Computing Conference (EDCC 2015)*, Paris, France, Sept. 2015a. URL `https://hal.archives-ouvertes.fr/hal-01211774`.

C. Sauvanaud, G. Silvestre, M. Kaâniche, and K. Kanoun. Data Stream Clustering for Online Anomaly Detection in Cloud Applications. In *11th European Dependable Computing Conference (EDCC 2015)*, Paris, France, Sept. 2015b. URL `https://hal.archives-ouvertes.fr/hal-01211774`.

C. Sauvanaud, K. Lazri, M. Kaâniche, and K. Kanoun. Anomaly detection and root cause localization in virtual network functions. In *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, pages 196–206, 2016. doi: 10.1109/ISSRE.2016.32. URL `http://dx.doi.org/10.1109/ISSRE.2016.32`.

G. Silvestre, C. Sauvanaud, M. Kaâniche, and K. Kanoun. An anomaly detection approach for scale-out storage systems. In *26th International Symposium on Computer Architecture and High Performance Computing*, Paris, France, Oct. 2014. URL `https://hal.archives-ouvertes.fr/hal-01076212`.

G. Silvestre, D. Buffoni, K. Pires, S. Monnet, and P. Sens. Boosting streaming video delivery with wisereplica. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XX*, pages 34–58. Springer, 2015a.

G. Silvestre, C. Sauvanaud, M. Kaâniche, and K. Kanoun. Tejo: A Supervised Anomaly Detection Scheme for NewSQL Databases. In *7th International Workshop on Software Engineering for Resilient Systems (SERENE 2015)*, Paris, France, Sept. 2015b. doi: 10.1007/978-3-319-23129-7_9. URL `https://hal.archives-ouvertes.fr/hal-01211772`.

C. Simache and M. Kaâniche. Measurement-based availability analysis of unix systems in a distributed environment. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 346–355. IEEE, 2001.

C. Simache, M. Kaâniche, and A. Saidane. Event log based dependability analysis of windows nt and 2k systems. In *Dependable Computing, 2002. Proceedings. 2002 Pacific Rim International Symposium on*. Citeseer, 2002.

Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 173–182, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835741. URL `http://doi.acm.org/10.1145/1835698.1835741`.

Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 285–294, 2012. doi: 10.1109/ICDCS.2012.65.

J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning*, pages 935–942. ACM, 2007.

C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *2010 IEEE Network Operations and Management Symposium - NOMS 2010*, pages 96–103, April 2010. doi: 10.1109/NOMS.2010.5488443.

Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto. Online failure prediction in cloud datacenters by real-time message pattern learning. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 504–511, Dec 2012. doi: 10.1109/CloudCom.2012.6427566.

A. W. Williams, S. M. Pertet, and P. Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007. doi: 10.1109/IPDPS.2007.370345.

W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629587. URL `http://doi.acm.org/10.1145/1629575.1629587`.

J. Zhang, M. Zulkernine, and A. Haque. Random-forests-based network intrusion detection systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(5):649–659, Sept 2008. ISSN 1094-6977. doi: 10.1109/TSMCC.2008.923876.

Y. Zhang, B. Hong, M. Zhang, B. Deng, and W. Lin. ecad: Cloud anomalies detection from an evolutionary view. In *Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on*, pages 328–334, Dec 2013. doi: 10.1109/CLOUDCOM-ASIA.2013.57.