Machine Learning Approaches to Early Fault Detection and Identification in NFV Architectures

Arij Elmajed, Armen Aghasaryan, Eric Fabre

Machine Learning Approaches to Early Fault Detection and Identification in NFV Architectures

Arij Elmajed
Nokia Bell Labs
Nozay, France
arij.elmajed@nokia.com

Armen Aghasaryan
Nokia Bell Labs
Nozay, France
armen.aghasaryan@nokia.com

Eric Fabre
Univ Rennes, INRIA
Rennes, France
eric.fabre@inria.fr

***Abstract*—Virtualization technologies become pervasive in networking, as a way to better exploit hardware capabilities and to quickly deploy tailored networking solutions for customers. But these new programmability abilities of networks also come with new management challenges: it is critical to quickly detect performance degradation, before they impact Quality of Service (QoS) or produce outages and alarms, as this takes part in the closed loop that adapts resources to services. This paper addresses the early detection, localization and identification of faults, before alarms are produced. We rely on the abundance of metrics available on virtualized networks, and explore various data preprocessing and classification techniques. As all Machine Learning approaches must be fed with large datasets, we turn to our advantage the softwarization of networks: one can easily deploy in a cloud the very same software that is used in production, and analyze its behaviour under stress, by fault injection.**

***Index Terms*—NFV, virtualization, network management, root cause analysis, fault identification and localization, Machine Learning, Max-Likelihood, fault injection, interventional learning**

I. Introduction

By virtualizing the network functions and decoupling them from hardware appliances and physical locations, the Network Function Virtualization (NFV) paradigm is a tremendous opportunity for enhancing networks operation. The possibility of developing Virtual Network Functions (VNF) using general-purpose programming languages and running them on commodity hardware provides a high level of flexibility and agility, as well as a faster time to market for new services and upgrades. It enables telecom operators to simplify, rationalize and develop a sustainable deployment of technical solutions.

In spite of its benefits, NFV also comes with an increase of complexity at different levels, and raises new challenges, especially for Network Management, where operators spend a tremendous amount of their time to ensure the reliability of the services offered to their customers.

One of the hardest challenges relates to fault and performance management. The identification of the cause, the location, and/or the type of network malfunctions that result in a degraded Quality of Service (QoS) becomes more complex in these new virtualized environments. For legacy networks, malfunctions are typically identified by metrics crossing a threshold configured by some field expert. This generates alarms which are supposed to pinpoint overload bottlenecks, faults, and failures. In reality, a malfunction generally results in cascades of correlated alarms, arising from different network layers, which renders the identification of the root cause problematic. Root cause analysis (RCA) and alarm correlation have been an active research topic for at least two decades. Virtualized networks make the problem even more difficult as more resources and additional layers are involved in the delivery of a service. But they also raise qualitatively new difficulties. The first one comes from the intrinsic dynamic architecture of such systems, which are designed to be constantly reconfigured. This is clearly an obstacle to both model-based and data-driven (learning-based) approaches to RCA. The second difficulty relates to the necessity to quickly identify a malfunction or a service degradation, possibly before metrics are severely impacted, as a quick detection and isolation is the key to adaptive resource allocation. Minimizing the human intervention and reducing the time to recovery are crucial within the paradigm of dynamic and adaptive networks realized through network virtualization. While several fault detection and localization solutions are available when the service performance and service-level agreement (SLA) are already jeopardized, here we will focus on early fault detection before an apparent QoS degradation takes place.

In this paper, we address two central challenges related to fault management in virtualized networks. One is the early detection of service degradation, before some metrics are severely impacted and trigger alarms. Our idea is that the *combination* of metrics, and in particular the way they jointly evolve, can be an early signature of a QoS degradation. The performance degradation sources we cover are related to typical resource shortages in virtualized environments, such as CPU overloads in virtual machines (VM), latency in messages exchanges, or overloads in disk access. The second challenge relates to the unavailability of models of the managed systems, although the (dynamic) topology might be accessible and usable. In fact, some of the RCA techniques rely on a system model and develop sophisticated diagnosis algorithms that exploit the model to infer the hidden fault states from observable variables. In dynamic virtualized networks such models are hard to obtain. In contrast, another family of techniques try to characterize fault situations by adopting a black-box approach, through a data intensive learning process. These so-called

978-1-7281-5684-2/20/$31.00 ©2020 IEEE

Machine Learning (ML) approaches require however labeled data which are also in general unavailable in sufficient volumes. ML requires labeled data as a starting point, which is generally a scarce resource as this means experts need to annotate logs of failure scenarios. Virtualized networks open a way around this bottleneck, as the true production software can be deployed in a controlled environment, where realistic perturbations can be injected. As we are not interested in severe failures but in yet "invisible" degradations, one can also imagine injecting such smooth perturbations in running/production services, which opens the way to reinforcement learning approaches that would adapt failure signatures to dynamic changes of architectures.

The contributions of this paper are as follows. We first propose a methodology based on the deployment of an experimental platform for virtualized network services enabled with the automated execution of resource perturbation schemes. As an example, we deployed ClearWater, a virtualized IP Multimedia Subsystem (IMS) services over OpenStack, fed with a traffic generator (SIPp), and executed series of resource perturbations (CPU, network latency, etc.) at different application nodes. Perturbation levels on the platform are carefully calibrated so that services are not yet impacted, and users do not yet start disconnecting. We then select the relevant metrics to collect on this platform, and generate datasets labelled with the injected perturbations. Raw data are not directly usable as they exhibit dynamics specific to the current execution, e.g. they can depend on the current load of the service (the specific number and activity level of users). We thus examine several data preprocessing techniques that avoid the model over-fitting while preserving the signatures of service degradation. Finally, we compare a family of classical ML approaches, including a statistical Max-Likelihood estimation, for the detection and characterization of the injected faults.

The paper is organized as follows. Section II presents the related work. Section III introduces the experimental setting, the stress injection approach and the data collection methodology. Section IV presents our pipeline for data preprocessing and feature selection. Section V presents the Maximum Likelihood approach and other ML approaches. Section VI evaluates the effectiveness of the proposed solutions; we compare our work to related research using data from different samples and with different traffic patterns.

II. Related work

There exist many RCA techniques for fault diagnosis which can be classified according to different aspects such as problem requirements, availability of data (size, form, velocity, etc.), or availability of a formal system representation (model).

Model-based diagnosis approaches take into account the system behavior and topology to construct an abstraction of the system in terms of a model on which the fault behaviours can be explained. Some contributions rely on building fault propagation models, in [1] authors build a behaviour model reproduced in different fault scenarios using the tile structure derived from Viterbi-based diagnosis algorithm. Sanchez et al. [2] use a self-modelling and multi-layered approach that captures dependencies between network services in Software Defined Network (SDN) and NFV context and propose a RCA module using network topology. Authors in [3] propose an approach based on Edge-Rank to find the key events that contribute to a fault and construct a dependency graph with the correlations among events and metrics, and a detection graph, to find a path of events that represents a fault. And in [4], they present a multi-level probabilistic inference model that captures the dependencies between different services in enterprise networks and describe an algorithm that uses this inference graph and service response times measurements for root cause localization.

Pham et al. [5] use a failure injection approach to construct a database with failure profiles, then with a Top-K Nearest Faults query algorithm they recognise a specific fault pattern and pinpoint the root cause. Authors of [6] use a statistical fault diagnosis, with Pearson correlation computed pairwise between neighboring nodes to detect anomalies. It is based on the fact that the Virtualized Network Functions (VNF) chain can be seen as a pipeline, where the input of the next network function of the chain is the output of the one before. These two works, only provide detection solutions without fault type identification and localization. One of the non-statistical data driven approaches uses fault injection techniques to generate data and applies a Random Forest (RF) algorithm [7] to detect malfunctions and localize the root cause (one task at a time) in the context of NFV. Although these two approaches [6] [7] achieve good level of accuracy, they do not provide any complexity analysis while using a high number of metrics. And, they do not address the model parameter optimization and do not check the robustness of the solution in presence of different load patterns.

For a joint data-driven diagnosis, authors in [8] combine ML with Granger causality for temporal correlations between anomalous values of Key performance indicators (KPIs) in order to construct a causality graph and localize anomalies based on a set of graph centrality indices. The authors show a high detection accuracy and low False Positives rate, but do not address the tasks of the localization and identification simultaneously. Authors in [9] propose a fault management framework for clouds, where they use Hidden Markov Models (HMM) and two ML clustering algorithms (K-Nearest Neighbor KNN and K-means) to construct a correlation-based performance models to detect and recognise anomalies. Authors in [10] propose an anomaly detection framework for OpenStack services. This work characterizes nominal operating modes of each process through clusters of K-means, and define a process anomaly as a significant deviations from the centroids of these clusters. The dependency graph of processes is then used to connect possibly faulty processes. In our work, we do not make use of the topology in the first place. Rather, we focus on the joint management of all processes metrics to detect and localize a malfunction before alarms are produced.

All these state-of-the-art techniques have proposed solutions for detection and localization separately and without fault type identification and when the system has already been affected by

faults. Some of these works use data from a specific network gathered over a long period of time and others used fault injection approach to generate their data. In our work, we target a fault injection based approach for early fault detection (binary classification), localization and identification.

III. Experimental platform for model learning

A. Stimulus-based approach

Faults are rare events which makes difficult to comprehensively learn their impact unless we are provided with rigorous specification of the system's faulty behavior. While such models are not available in cloud-based networks, obtaining sufficient volumes of labeled data through passive network monitoring is not realistic neither; this would imply observing the system for a long time to encounter all types of these rare events. For these reasons, we choose an interventional approach which consists in provoking the faulty situations of interest and "shortening" the waiting time to their natural occurrence. More specifically, we apply the stimulus-based approach [11] which consists in methodically applying a series of elementary resource perturbations in each node of a distributed virtualized application with the purpose of elucidating causal dependencies between local and remote states. Here, we aim at discovering other types of models (statistical and ML models), different from [11], however we will use the same stimulus-based approach for a controlled data acquisition.

The solution is to set up an NFV based application in a private managed cloud environment and to learn its behaviour through resource perturbations without interfering with other applications; this environment is called "Sandbox"; in the present case it is an NFV-based Sandbox. The application nodes are taken as communicating black-boxes each one relying on a limited set of computing resources: CPU, memory, disk Input/Output (I/O), network I/O, etc. Given an input load, the behavior of an application is then essentially determined by the quantity of computing and communication resources consumed by the nodes. We consider therefore any types of faults that impact the computing and communication resources.

We can see the stimulus as a button that we trigger to have a system fault each time. In our case, due to the limited access to the underlying hardware layer, we consider only three perturbations categories (CPU, Disk I/O, or Network I/O). The considered faults categories are:

1) CPU stress that means increasing the CPU usage in the VM using the stress-ng tool[1];
2) Disk stress, where the disk usage increases by using the stress-ng tool;
3) Network stress, in which the network delay increases using the open source tool tc [2].

B. Test-bed architecture

To generate faults for our solution, we used a test-bed platform. First we present the telco application which is installed in an open cloud software, OpenStack [3], where the application nodes are IMS components from ClearWater[4] open source project depicted in Fig. 1. The ClearWater IMS is an implementation of the IMS core standard [12]. IMS functions

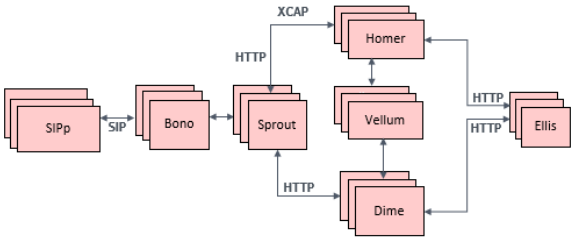

Fig. 1. ClearWater open source project components

are implemented in VMs and are taking full advantage of cloud computing technology.

- Bono: it is the first point of entry in the IMS core network. It is responsible for routing incoming SIP transactions to the IMS registrar server (the Proxy CSCF);
- Sprout: it is a combined SIP registrar and authoritative routing proxy;
- Dime: it provides two functions:
  1) Homestead: provides web interfaces to Sprout for retrieving authentication credentials and user profile information;
  2) Ralf: provides an HTTP API that both Bono and Sprout can use to report billable events.
- Vellum: it is used basically to store authentication credentials and profile information;
- Homer: it is a standard XML Document Management Server (XDMS) used to store multimedia telephony service settings documents for each user of the system;
- Ellis: it is a web provisioning portal to create subscribers;
- SIPp [5]: it is a Load generation tool; creates multiple SIPp instances each of which emulates a pair of users. It performs actions in control-plane only. At first we have ramp-up period of 10 minutes to set-up subscribers load. Each couple of the created subscribers will try to register or update the registration every 5 minutes. Then it will attempt to send an INVITE with a default probability of 16%, and RE-REGISTER with a probability of 16%.

Our platform depicted in Fig. 2 is composed of several functional blocks:

- Fault catalog: which is the list of perturbations that we can emulate on the platform;
- Ansible[6] orchestrator: which enables us to manage and execute remotely the perturbations in our platform. The tests that are executed are organized in a playbook, that are called "perturbations";

[1]https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html
[2]https://wiki.debian.org/TrafficControl
[3]https://docs.openstack.org/ocata/fr/install-guide-ubuntu/index.html
[4]https://www.projectclearwater.org/
[5]http://sipp.sourceforge.net/
[6]https://www.ansible.com

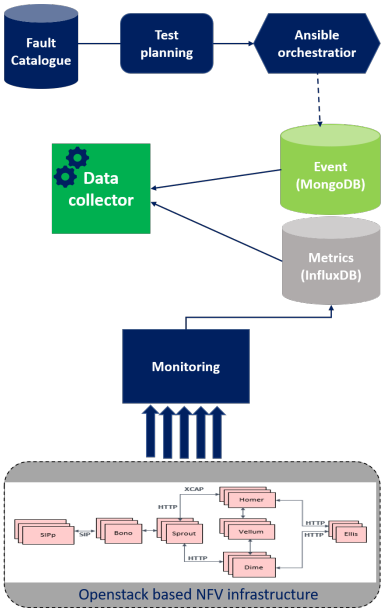

Fig. 2. Test-bed architecture

- OpenStack based cloud infrastructure: is our ClearWater virtual IMS platform;
- Monitoring tools:
  1) InfluxDB[7] is a time series based database for storing the metrics collected;
  2) Telegraf[8] is the SNMP agent used for collecting the metrics in our VMs;
  3) Grafana is our dashboard that we configure to display the graphs for the metrics that we are collecting.
- Event storage: is a MongoDB[9] database used to store the stimulus start and stop, such a stimulus or a perturbation is considered as an event with metadata indicating stimulus name, stimulated metric, and resource type.

C. Fault injection calibration

In contrast with [11], we are interested in early fault detection, and therefore the perturbations need to be calibrated in a way to not push the system into an outage mode. The calibration of the injected perturbations is crucial, and we performed numerous tests to select the appropriate levels. Our observation is that too severe perturbations lead to a collapse of the system: clients start being disconnected and never manage to reconnect again, so that the metric "connected users" goes to zero or stabilizes at a low value. By contrast, too small perturbations could have a non-significant influence on metrics. We thus selected a single perturbation intensity for each experiment, slightly below the collapse level, and guaranteeing at the same time that all collected metrics had stationary behaviors.

[7]https://www.influxdata.com/time-series-technical-paper-2/
[8]https://www.influxdata.com/time-series-platform/telegraf/
[9]https://www.mongodb.com

We have generated perturbations with three different traffic patterns: the first one is regular traffic pattern (e.g. day traffic) with 50k subscribers, the second one represents peak periods with a 100k subscribers, and the third one with more INVITE transactions 85% and REGISTER rate of 15%. We have six different datasets: four datasets are generated with the regular traffic pattern, one with the high peak, and another one with more INVITE transactions.

D. Collected metrics

In Machine Leanring, data collection is the most critical step for building the desired knowledge, about the faulty behaviour in our context. We collected two types of metrics, some in the virtualization plane, in the form of resource-related metrics (CPU, Disk, memory, network, processes), and other in the application layer, mostly QoS metrics (latency, number of incoming INVITE/REGISTER). As the application provides numerous QoS metrics, we had to choose the most appropriate ones the learning purposes. Stationarity is an important criterion for metric selection, as the statistical properties should not change over different periods of time in a given perturbation scenario. Often, metrics come in a cumulative form (e.g. number of packets sent), then one has to replace them by their derivatives to collect instantaneous information and ensure independence of samples. Several metrics "duplicate" others, or are highly correlated. We took a single version of them. Finally, we observed periodic behaviors in data due to the scenarios executed by the traffic generation software (see Fig. 3). We rejected this nuisance by sampling our data along the different phases of these scenarios, at the expense of augmenting variance.

We end up with around 63K rows of data with 35 features[10]. Each line is thus a pair (x, y) where the feature part $x = (x_0, x_1, ..., x_{34})$ gathers our metrics, and the fault class y can take 13 possible values: either the no fault or a pair <fault-type, fault-location>:

$$\begin{aligned} y \in \{&Normal, CPU_{Bono}, CPU_{Dime}, CPU_{Sprout}, \\ &CPU_{Vellum}, DISK_{Bono}, DISK_{Dime}, \\ &DISK_{Sprout}, DISK_{Vellum}, Network_{Bono}, \\ &Network_{Dime}, Network_{Sprout}, Network_{Vellum}\} \end{aligned}$$

IV. Data preprocessing

The early detection of performance degradation in metric vectors requires a careful analysis of the available data, to select the right metrics, accommodate nuisances like transient behaviors, outliers, or the effect of exogenous system load. It also requires a rigorous experimental methodology, to avoid biases that could impact detection and classification performances.

[10]Data is available upon request

A. Necessity of stationary data

Our fault injection approach consists in perturbing one resource on a virtual machine, at an appropriate level of severity, and in collecting metrics over a rather long period of time. We assume the collected data are stationary, in order to collect a large number of hopefully independent signatures of a given perturbation, which then enables a statistical analysis or ML methods. Too severe perturbations will produce severe service degradation, resulting in metrics either drastically increasing or decreasing. For example, the number of connected users may go to zero, with all traffic metrics. Conversely, too weak perturbations will be non-detectable. Fig. 3 illustrates

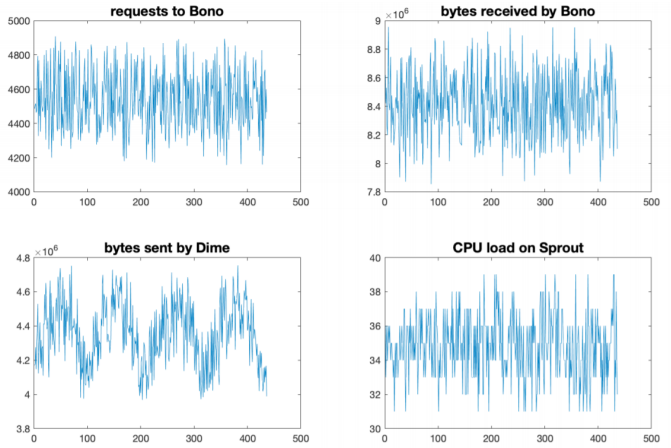

Fig. 3. Samples of available metrics.

samples of some collected metrics, related to numbers of requests, bytes received, bytes sent (per second), and CPU load. One can observe that several metrics approximately follow a Gaussian distribution, as witnessed by Fig. 4. Nevertheless, their dynamic ranges are extremely different, which requires a re-normalization of data if we want to handle them jointly. Not all metrics have such nice behaviors. First, some of them come in cumulative form (e.g. total number of bytes received), so they need to be handled under their derivative form (see also the discussion about experimental biases). Then, there are numerous outliers, so in our experiments we have discarded data that were beyond 3 times the standard deviation above/below the mean value. Other metrics, like the CPU load measurements (bottom right in Fig. 33), are quantified at the precision of 1%, and exhibit little fluctuations around their mean, hence the Gaussian approximation for them is far from perfect. We have also observed metrics with spiky behaviors (typically, variations of buffer loads) for which the Gaussian approximation would clearly be inappropriate. Despite the almost Gaussian behavior of most metrics we have selected, some of them exhibit periodic pattern. This is quite visible on the Dime and Vellum metrics (bottom left in Fig. 3). This is due to a 30' periodic behavior of the SIPp load generator that plays registration and call scenarios over its database of users. Our data range over 2 hours, hence 4 periods observed in the data. We have chosen to ignore this artificial alternation of high and low activity levels, and considered averaged behaviors over the 2 hours, at the expense of a larger variance estimate on these data. Similarly, we have also noticed a sub-period of 5' in several metrics, still due to the SIPp activity, that we have also chosen to average out.

Finally, in some experiments, some metrics are either unmeasured or almost constant. We have chosen to ignore them when this is the case. For example, too strong perturbations on the CPU of VMs can result in almost saturated metrics slightly below 100%. We consider that this can be easily detected by a direct observation of the metric. We are rather interested in weaker perturbations that would not trigger an alarm, and for which a joint analysis of several metrics would be necessary to detect the fault. As another example, we have also observed that latency perturbations in VM communications could result in a drastic modifications in the statistics of some single metrics such as the size of cached data in memory. Again, this is then easily detectable, so we have chosen to ignore such metrics for the detection of latency perturbations.

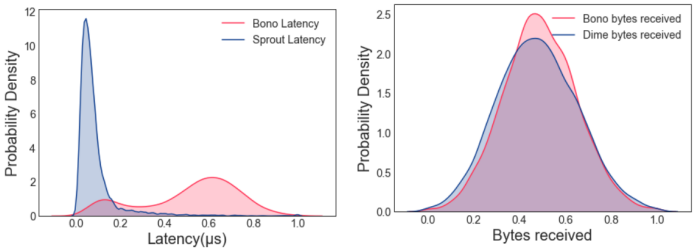

Fig. 4. Left: probability distribution for measured latencies at Bono and Sprout. Right: normalized probability distributions of bytes received per second, by Bono and by Sprout.

B. Data normalization

The collected metrics have extremely different dynamic ranges, both due to the nature of the measured quantities (bits/s, requests/s, CPU load), and due to the current load of the system. We want to study how metrics jointly evolve in case of a silent perturbation, without favoring one or another. We also aim to be insensitive to the "nuisance parameter" represented by the number of users present in the system. Therefore, in order to get comparable dynamics, insensitive to system load, we examine two techniques of normalization of the selected metrics.

The first normalization technique assumes the data follow a Gaussian distribution; we center and normalize them in variance. In doing so, the average activity level is set to zero (so all information lying in the metric averages is ignored), and only the correlation between metrics will be analyzed. This is equivalent to considering the so-called Z-score of data. Formally, let μ and σ^2 be the mean and variance of some collected metric over 2 hours, we replace each value x by $(x-\mu)/\sigma$.

The second normalization technique is slightly less brutal. It simply consists in resizing the data so that the minimal (resp. maximal) value of each metric is set to 0 (resp. 1). Formally,

let x_{min} and x_{max} be the observed minimal and maximal values of x over our 2 hours experiment, we replace each x by $(x - x_{min})/(x_{max} - x_{min})$. By contrast with the previous one, this normalization may preserve information about the mean and variance of the metric.

C. Metric selection

Which metrics to consider, and how many of them should we use for early fault detection? Several criteria come into play. First of all, we select the metrics directly related to the activity level of both the application (ClearWater) and the virtualization layer (OpenStack), such as the bytes received and sent by software components, the numbers of requests, the CPU load of a VM, etc. We keep also, the metrics reflecting the health state of the entire system (latency measurements, numbers of request failures, etc.). As mentioned above, it is important that these metrics exhibit stationary behaviors. Another sanity check is that they react to the injected failures. Fig. 5 illustrates the correlation patterns of the 21 selected metrics (from 35 gathered in the beginning), for two different perturbations. One can observe that the joint behavior of these metrics is a signature of the injected fault, and that this information is preserved by the normalization of data.

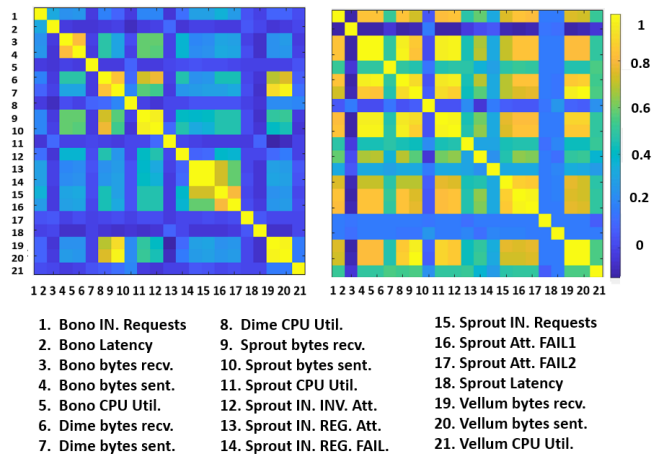

Fig. 5. Correlation matrices of 21 metrics for two injected perturbations: a CPU load on Bono (left), and a latency in communications on Dime (right).

We have performed experiments on the classification of 12 faults (3 fault types over 4 components), using the Extreme Trees Classifier, fed with z-score data. Fig. 6 sorts selected metrics according to their influence in the classification decision. When some of them (the top 17) are simultaneously used, the classification accuracy is almost perfect.

D. Risks of experimental biases

Some metrics exhibit non-stationary behaviors and their successive samples are not independent (e.g., see Fig. 7, left). We have chosen not to model this memory effect, and assume independence of samples. Our experimental data thus collect samples over a 2 hours period for each injected perturbation. The data for training and testing belong to different samples

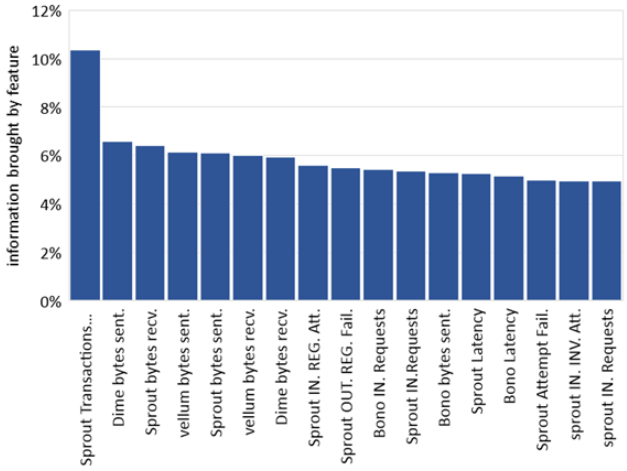

Fig. 6. Gini influence of metrics for the XTC classifier.

generated in different period of time. If metrics with memory are used, they can reproduce some kind of timing information, that would sign the moment at which data were collected. This information remains even if data are normalized. We have performed experiments with fake metrics, corresponding to samples of a Brownian motion, centered and re-normalized in variance (see Fig. 7, right). As the number of such fake metrics augments, the fault classification performance increases, and is almost perfect with simply 10 Brownian motion samples. Our interpretation is that these synthetic metrics with memory jointly form a time vector that uniquely identifies the moment at which experimental data were collected. It is therefore crucial to reject non-stationary metrics, that exhibit a memory effect.

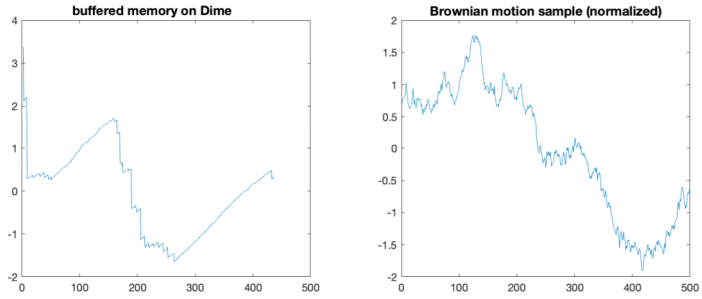

Fig. 7. Left: a metric with memory (buffered memory on Dime). Right: a forged fake metric, as a Brownian motion sample.

V. Approaches for Early Fault Localization and Identification

We choose four different approaches. We first handpick Random Forest (RF) because several previous contributions reported a good accuracy in detecting and identifying service degradations. We want to check whether RF still performs well for soft degradations, where one only collects preliminary symptoms. The second selected approach is Extreme Gradient Boost (XGBOOST) as one of the most popular recent classification techniques. It solved many complex problems, and

many Kaggle competitions were won using this algorithm. The third one is a classical statistical approach, Max-Likelihood algorithm, assuming metrics follow a Gaussian (multi-valued) distribution. Max-Likelihood exploit the correlation patterns between features/metrics which supposedly form different signatures for our fault classes. It provides the minimal error classifier, if the collected samples are independent and follow a Gaussian law. Finally, the fourth method, K-Nearest Neighbors, it is chosen because it is suitable for our data velocity and was proven to achieve a good accuracy in similar problems. For the implementation, we use Python scikit-learn library for RF, KNN and XGBOOST, and Matlab for Max-Likelihood.

A. First approach: Random Forest

The general idea is to rely on an ensemble of decision models to improve accuracy. RF [13] is a supervised learning algorithm that generates and merges several decision trees into one forest, which enhance the accuracy of the model. For each decision tree in the forest a random sample of data is chosen and at each split a subset of features are randomly chosen to avoid over-fitting and minimize multicollinearity. We find the best split point and split it to daughter node using a method for best split (Gini index or entropy) until we reach the nodes that are too small to be split called leaves. Then when we get all the trees, we classify using the highest voted class among all the estimators.

For the RF approach, we need to tune the best hyper-parameters (The parameters which can be defined by the user before starting training the model), in particular the number of estimators (trees) and the maximum depth for a tree ; because more trees imply more computational costs, and the improvement is minimal after a certain number of trees, so we need to find the optimal number. We compare between two algorithms as shown in Fig. 8 : Grid search algorithm and Random search [14].The first method tries every single combination of hyper-parameters from a range of values and the latter simply tries random combinations of hyper-parameters from a range of values to find an optimal solution for a model through Cross Validation (CV), avoiding over-fitting. We choose K-fold the most popular CV method with k=3 (k: the number of groups to which the data will be split). For the implementation, we set 20 iterations, each one trying a new combination of hyper-parameters: We choose to use the

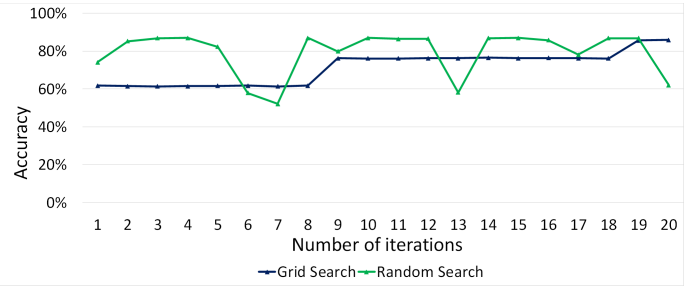

Fig. 8. Comparison between two hyper-parameter tuning approaches

Random search algorithm because it has a better accuracy and a faster convergence. The optimal values obtained are 29 for the maximum depth of a tree and 872 for the number of estimators in the forest. It should be noted that RF preprocessing is not a necessary step, as this algorithm is less sensitive to data scaling then other approaches.

B. Second approach: Extreme Gradient Boost

The Extreme Gradient Boosting (XGBOOST) [15] is also a supervised ML algorithm, but instead of training models separately like RF, it trains sequentially, it has a more regularized formalism to avoid over-fitting; The first weak tree based classifier uses all the observations, it computes the residuals (the difference between the predictions and the actual values) and updates weights for incorrectly predicted observations. Then, for the second decision tree the observation with the maximum weights will be selected to construct the model. We will apply the same procedure by modifying weights for wrongly classified samples. So, we will iterate sequentially until constructing the n-th decision. Finally, all of these weak learners are combined to get a strong weighted prediction.

C. Third approach: Max-Likelihood classification

The Max-Likelihood classification is a classical statistical approach with sound mathematical foundations for its decision rule. We assume that all the features are normally distributed and we estimate the probability that a given observation belongs to a specific class; the class y_i with the highest computed likelihood is assigned to the observation X.
Each sample will be associated with a class (a concatenation between type and localization). We implement Logarithmic Max-Likelihood (ℓ) : a supervised classification by calculating the following function for each observation vector in the data-set :

$$\begin{aligned} & \text{Log-likelihood} \\ = \quad & (X-\mu_i)^T\Sigma_i^{-1}(X-\mu_i) + log(det(\Sigma_i)) + N_i log(2\pi) \end{aligned}$$

where μ_i is the mean and Σ_i the correlation matrix of random vector X in class i (recall that data are normalized, so the covariance is actually a correlation), and N_i the dimension of the data in class i.

The proposed approach was initiated by the visualization of the correlation matrices of each class y_i. It is observed as expected that the correlations between features change depending on the classes. We take the example of features dependencies difference between two classes : CPU_Bono, Network_Dime. The blocks of features are highly correlated in the data Network in Dime perturbation around one value and for the CPU Bono the blocks correlation are around 0.5 as depicted in Fig. 5.

D. Fourth approach: K-Nearest Neighbors (KNN)

KNN [16] is a supervised non-parametric ML algorithm used for pattern recognition and intrusion detection systems; it is based on feature similarity measure. Given N training vectors, it identifies K nearest neighbors for vectors, then using

a voting system it finds the class to which a given vector will be associated. We choose to use this approach since our dataset is not so voluminous after the removal of noisy features. One crucial step when using KNN is to choose parameter K, if K is too low, it will bias the results and if it is too high it will increase complexity. So we tuned the hyper-parameter to find the optimal K value using Randomized search and we found that best value is 15 nearest neighbors.

VI. Results and comparison of approaches

We show some experimental results to demonstrate the effectiveness of our approaches. We first tried to train models with 70% for train and 30% test, but it seems to be not realistic and not very robust methodology for that reason we choose to train our models on a mixture of datasets from different traffic patterns execution with regular traffic pattern (50k subscribers), pattern with higher ratio of INVITE (85% ratio of Invite/Register) and high traffic pattern (100k subscribers). And then the test was carried out on the dataset obtained from a distinct execution with a regular traffic pattern.

A. Fault detection

We detect here whether a fault has occurred or not by recognising preliminary symptoms; it is a binary classification. As shown in Table I, we compare between the four described approaches using three performance indicators: Precision, Recall and AUC score (Area Under Curve) can be understood as a measure of the probability that the model will classify a random positive example over a random negative example:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

where TP : True Positive, FP: False Positive, and FN: False Negative.

TABLE I
Detection performance with different ML algorithms

Algorithms/Performance	Precision	Recall	AUC score
RF	93%	94%	99%
XGBOOST	93%	93%	91%
Max-Likelihood	98%	98%	84%
KNN	89%	89%	96%

All approaches have a very good Precision and Recall around 90%, and away from traditional ML algorithms, Max-Likelihood classifier performs also as good as the other approaches. Although, it is important to assess the binary classification using AUC score, since it tests the efficiency of a binary classifier on average across a range of possible decision thresholds. The results obtained with RF, XGBOOST and KNN with an AUC score above 90% that means that the three algorithms are able to distinguish between normal and faulty classes, Max-Likelihood AUC score is 84% less than the top three classifiers have also a good discrimination capacity to distinguish between classes with faults and normal. So it is an easy task for the ML algorithms to recognise faulty patterns and normal behaviour especially if the preprocessing part is done correctly.

B. Identification and localization of faults

For the fault localization and identification task, we will evaluate the classification performance between the described approaches using Precision, Recall and F1-score since it is a harmonic mean of Precision and Recall and it is useful to evaluate if we have imbalanced classes (our samples per class are not even).

TABLE II
Fault localization and identification performance with different ML algorithms

Algorithms/Performance	Precision	Recall	F1-score
RF	82%	82%	81%
XGBOOST	81%	81%	80%
Max-Likelihood	44%	43%	43%
KNN	73%	73%	72%

As illustrated in table II, the two highest performance values belong to RF and XGBOOST with 82% and 81% of Precision and Recall. Then, we have KNN with a Precision, Recall of 73% and a F1-score of 72%, it has a performance which can be enhanced, since KNN is an algorithm that can suffer from a phenomenon called "dimensionality curse": data dimensions can affect badly algorithm performance. Max-Likelihood classifier have the worst performance, that can be explained by the data size. There is a huge gap between results obtained for detection using Max-Likelihood classifier and results for fault identification and localization. It is interpreted by the fact that some fault patterns are quite similar and this can lead to prediction errors. For the four approaches, Precision and Recall are equal, meaning that an equal amount of users has been classified as False Positives and as False Negatives. For the learning time, the RF took 59 second since we are using 872 trees, 24 second for XGBOOST, 3 second for KNN and the longest one was Max-Likelihood with 1 minute and 40 seconds.

We take a closer look to the classification accuracy for each class using confusion matrix.

Fig. 9 depicts the confusion matrix of the RF classifier. Lines represent injected faults, columns stand for detected faults. We first have CPU faults (top left), then Disk faults (center), and Network faults (bottom right), and within each block of 4, locations range over components Bono, Dime, Sprout and Vellum. On the diagnonal, and below percentages, quantities A/B stand for the number of cases correctly recognized over the total number of cases. Off the diagonal, the single number A stands for the number of wrong classification (for the same B, constant on a line).

RF classifies well almost all the classes except the latency in Vellum and Dime; this is explained by the fact that the two perturbations have a close pattern and a similar impact

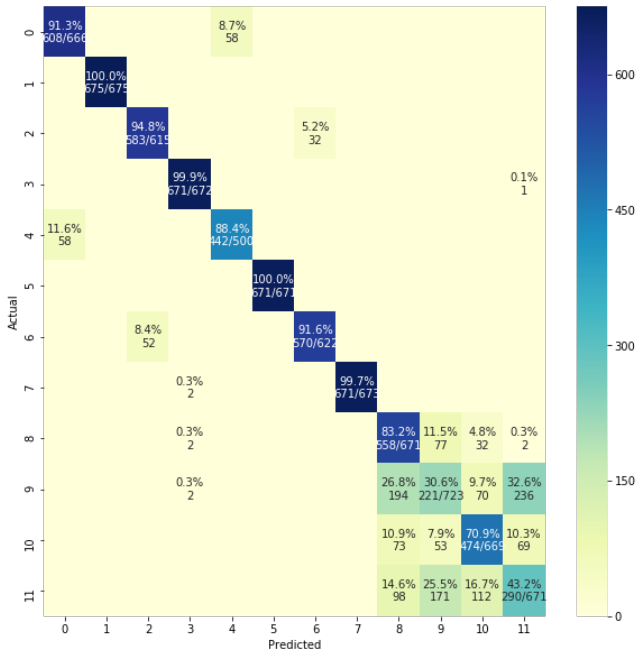

Fig. 9. Confusion matrix for RF

on our NFV platform, since Dime and Vellum are two highly correlated VNFs in the platform. The RF is the best classifier for this task since number of trees was tuned to find the optimal value, but with a large computational cost during runtime as a drawback.

XGBOOST and RF has almost the same accuracy for each class and the same misclassified classes, which can be explained by the fact that the two approaches are tree based algorithms.

VII. Conclusion and Future Work

This work demonstrates that the joint use of performance metrics enables the early detection, identification and location of faults in virtualized networks, before QoS is impacted. One can train classifiers to recognize these fault signatures by injecting gentle perturbations in the network, at levels that do impact too much services and nevertheless enable the learning techniques to grab sufficient information.

We have trained our ML models on a mixture of datasets obtained with different load levels and protocol usage patterns, in order to increase robustness to nuisance parameters. The evaluation was performed on a distinct dataset with a fixed load level. RF and XGBOOST were best to recognize different types of fault injections such as CPU exhaustion, excessive disk I/O, and network latency. They show excellent scores and resist well to different traffic patterns. This suggests that such approaches combining gentle fault injection and learning could be deployed online, in production services.

Despite these early promising results, we need to further investigate the robustness properties and the generalization of the proposed methods. For example, the selection of metrics, the necessary preprocessings (e.g. ouliers removal) and normalization of data, the calibration of injected faults, the resistance to traffic loads or to user profiles are important to get good classification levels, and still need further investigation. We also assumed that all collected samples were independent, while Markov models could possibly better account for memory phenomena in such systems. Nevertheless, the present work underlines that the joint behavior of multiple continuous metrics is already quite informative, as opposed to a more traditional per-metric alarm generation approach.

We plan to extend our work by building another statistical model to express the correlation between VNFs and resources, by exploiting the topological structure of the relationships between the VNF components. We also plan to use this stimulus-based approach on a Docker based platform and test our model.

References

[1] A. Aghasaryan, C. Dousson, E. Fabre, A. Osmani, Y. Pencole, "Modeling Fault Propagation in Telecommunications Networks for Diagnosis Purposes," World Telecommunication Congress, September 2002.

[2] J.M. Sanchez Vilchez, I. Grida Ben Yahia, N. Crespi, "Self-Modeling based Diagnosis of Services over Programmable Networks," 2nd conference on Network Softwarization (Netsoft2016), June 2016.

[3] Q. Zhu, T. Tung, Q. Xie, "Automatic Fault Diagnosis in Cloud Infrastructure",IEEE International Conference on Cloud Computing Technology and Science, 2013.

[4] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D.A. Maltz, M. Zhang, "Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies," SIGCOMM'07, August 27–31, 2007.

[5] C. Pham, L. Wang, B. Chul Tak, S.Baset, "Failure Diagnosis for Distributed Systems using Targeted Fault Injection," Transactions on Parallel and Distributed Systems 2016.

[6] D. Cotroneo, R. Natella, S. Rosiello, "A Fault Correlation Approach To Detect Performance Anomalies in Virtual Network Function Chains," IEEE 28th International Symposium on Software Reliability Engineering, 2017.

[7] C. Sauvanaud, K. Lazri, M. Kaaniche, K. Kanoun, "Towards Black-Box Anomaly Detection in Virtual Network Functions," 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, 2016.

[8] L. Mariani, C. Monni, M. Pezze, O. Riganelli, R. Xin, "Localizing Faults in Cloud Systems," IEEE 11th International Conference on Software Testing, Verification and Validation 2018.

[9] B. Sharma, P. Jayachandran, A.Verma, C.R. Das, "CloudPD: Problem Determination and Diagnosis in Shared Dynamic Clouds," IEEE 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, 2013.

[10] T. Niwa, Y. Kasuya, T. Kitaha, "Anomaly Detection for OpenStack Services with Process-Related Topological Analysis," 13th International Conference on Network and Service Management, 2017.

[11] A. Aghasaryan, M. Bouzid, D. Kostadinov, "Stimulus-based Sandbox for Learning Resource Dependencies in Virtualized Distributed Applications," 20th Conference on Innovations in Clouds, Internet and Networks (ICIN) 2017.

[12] 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; IP Multimedia Subsystem (IMS); Stage 2 (Release 16)

[13] L. Breiman, "Random Forests," Machine Learning, 45(5–32), 2001.

[14] J. Bergstra, Y. Bengio, "Random Search for Hyper-Parameter Optimization", Journal of Machine Learning Research 13 (2012) 281-305.

[15] T. Chen, C. Guestrin, "XGBoost: A Scalable Tree Boosting System," KDD '16, August 13-17, 2016.

[16] P. Cunningham, S.J. Delany, "k-Nearest Neighbour Classifiers," Technical Report UCD-CSI-2007-4, March 27, 2007.