Unsupervised Anomaly Event Detection for Cloud Monitoring using Online Arima

Florian Schmidt*, Florian Suri-Payer*, Anton Gulenko*, Marcel Wallschläger*, Alexander Acker* and Odej Kao*
*Complex and Distributed IT-Systems Group, TU Berlin, Berlin, Germany
Email: {firstname.lastname}@tu-berlin.de

***Abstract*—Virtualization offers cost efficient usage of digital resources. Thus, dedicated hardware solutions are transferred into virtualized services running in the cloud. Such softwarization of hardware is for example the IP multimedia subsystems, which telecommunication system providers currently move to the cloud. The dedicated hardware solutions provided a reliability of 99.999% in the past, but the virtualized services come with higher complexity due to the fragile computation stack and cannot provide such high requirements. Zero touch administration of such fragile systems can help to detect automatically anomalies, find root causes and execute remediation actions. This work focusses on the detection of degraded state anomalies. We propose an unsupervised detection approach using the Online Arima forecasting algorithm using real-time monitoring data. This approach is evaluated on a testbed running an open source implementation of the IP multimedia subsystem (Clearwater) executed on a replicated Openstack cloud environment. Results show the applicability of the Online Arima based anomaly detection with high detection rates and low number of false alarms.**

***Index Terms*—anomaly detection; service reliability; NFV**

I. Introduction

Virtualized infrastructure is commonly used within large data centres in order to provide flexible and scalable services to customers. Through the logically uncoupled virtualization from the physical machines, virtual machines can be used to scale-out and migrate services throughout their physical location of the infrastructure. This is also known as cloud computing and gains popularity especially for telecommunication providers, which softwarize their dedicated hardware solutions in order to benefit from the flexibility and cost-effectiveness. For telco-providers, the given dedicated hardware components provide a reliability of 99.999% [1]. Due to the complexity of the computation model, softwarized components cannot cope with the high demand of reliability. The fragility of such systems causes the usage of system administrators, maintaining the continuous operation of the services [2].

Active and passive monitoring tools for measuring the system's performance are widely used by administrators. Based on these measurements, alerts are triggered by expert-based fixed thresholds in order to inform about failures and anomalies. Due to system load changes, these thresholds are difficult to maintain as these must be set dependent on the individual service or operating system's resource usages in time. Thus mostly, the values are not changed frequently and may just provide information about problematic hosts, when it is already too late to remediate those. Finding an anomaly, before it causes a component to fail, is a challenging task as it is highly dependent on the system's individual behaviour. Human administrators need therefore a lot of time to investigate large amount of historic data to gain detailed insights to find the actual root cause. Then, they are able to start good remediation counter actions, keeping the system highly reliable. Self-healing pipelines are proposed to automatically detect and remediate anomalous systems, helping current administrators to detect more efficiently problems and give recommendations to select accurate counter actions [3].

This paper introduces therefore an unsupervised anomaly event detection approach for automated prediction of abnormal system changes. Thus, we like to detect anomalies before components fail. The approach is based on the forecasting algorithm Online Arima [4], which is used for learning the normal system behaviour of an individual component of interest. We assume that the system runs most of its time in a normal state and anomalies happen rarely. Consequently, we iteratively learn each monitored data point. The basic idea of our approach uses this model to predict the next expected monitoring data point and compares it with the actual measured values. As the model should be able to predict more accurate normal behaviour, the difference between those values should be smaller than those of anomalies. These differences will be detected and recommended to the administrator. As we monitor individual components, the approach is scalable for distributed systems and the administrators gain access to the individual component behaviours. Furthermore, we concentrate in this work on the service layer and provide an evaluation based on a testbed running an open source implementation of the IP multimedia subsystem executed on a replicated Openstack cloud environment.

The rest of the paper is structured as followed. First, we present related approaches capturing the state of the art of further anomaly and state change detection methods. Section III describes the background of the Online Arima algorithm, which is used within our approach. Section IV describes our unsupervised, anomaly detection approach. Section V shows the evaluation setup capturing the infrastructure testbed and analysis pipeline. Section VI describes and discusses the results. Lastly, Section VII concludes this paper and gives an outlook for future work.

978-1-7281-0359-4/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC-Companion.2018.00037

II. Related Work

Anomaly detection is a long standing and widely researched area with applications in diverse domains. The proposed approaches can be grouped into the two different categories, supervised and unsupervised methods.

Supervised approaches use labelled system information to train the machine learning models. Basically, the anomaly detection algorithms use either classification or regression models trained by data containing the information whether the data point is an anomaly or not.

Sauvanaud et al. [5] examine in their work a supervised ensembler approach for anomaly detection. They combine several different supervised classification algorithms, performing anomaly detection for individual services by using weighted voting mechanism. Thus, the collection of algorithms predicts whether a component is normal or not. The evaluation is based on a Virtual Network Function (VNF) scenario similar to our setup as presented in this work. Also, Liu et al. [6] propose a supervised anomaly detection strategy. As bases, they use the concept of self organizing maps (SOM). The SOM is configured to represent the whole infrastructure in order to predict the overall performance of the system. Based on this model, they show that anomalies within virtual machines can be detected in related regions of the infrastructure. In contrast to our work, we do not aggregate data from several hosts in a single model, but create models per individual host in order to capture its own behaviour.

As supervised approaches need labelled training data, those are mostly insufficient to be provided by expert-labelling as the manual labelling is too time consuming. Automatic creation of data sets assume, that one can collect training data from a running system. This would potentially harm the productive system and is therefore undesirable. For practical usage, unsupervised methods are therefore investigated, having the positive properties of performing the same task, but not using labelled input data. Thus, our work aims also an unsupervised detection technique.

Dean et al. [7] are also using SOMs, but propose an unsupervised detection model. They aim to predict failures within a cloud infrastructure, but do not consider degraded state anomalies as use case. In contrast to these deeplearning models, Cotroneo et al. [8] use simple statistical correlation analysis between system components in order to automatically detect anomalies within an NFV use case. Changes within these correlation are recommended as anomalies. Again, the focus of Cotroneo et al. is to combine data from different hosts. This approach suffers of scalability issues, which dynamic infrastructure introduces. Thus, we aim to create unsupervised detection models for individual hosts, which can be applied directly on the monitored entity.

III. Online ARIMA

A common approach to time series analysis is the Autoregressive Moving Average Model (ARMA) [9] [10]. Assuming that the time series is complete, i.e. there is no missing data, the ARMA model can supposedly identify the latent data correlation, allowing the forecast of future values.

Liu et al. [4] propose a game theoretic framework for online learning to forecast values. In their setting, an online player sequentially commits to a decision (forecasting the next data point) and consequently suffers a loss (prediction error). They assume coefficient vectors (α, β) that are set by an adversary and are at no time disclosed to the learner. Additionally, since the learning part has no ability to infer the actual noise term ε_t of a time step, this too is generated by the adversary, while remaining undisclosed to the learner.

At time t, the learner predicts a value $\tilde{X}_t$ and learns the true value X_t, as we refer to X_t as the data received at time t, subsequently suffering a loss l_t:

$$l_t(X_t, \tilde{X}_t) = l_t(X_t, \nabla^d \tilde{X}_t + \sum_{i=0}^{d-1} \nabla^i X_{t-1}) \tag{1}$$

Effectively, the original model ARIMA(k, d, q) is to be approximated by another model ARIMA($k+m$, d, 0). The noise terms are dropped and attempted to be compensated by extending the number of lags to regress upon by m. A new $(k+m)$-dimensional coefficient vector γ weights past observations, formulating the model:

$$\tilde{X}_t(\gamma) = \sum_{i=1}^{k+m} \gamma_i \nabla^d X_{t-i} + \sum_{i=0}^{d-1} \nabla^i X_{t-1} \tag{2}$$

and subsequently the loss:

$$l_t(X_t, \tilde{X}_t) = l_t(X_t, \sum_{i=1}^{k+m} \gamma_i \nabla^d X_{t-i} + \sum_{i=0}^{d-1} \nabla^i X_{t-1}) \tag{3}$$

Liu et al. present two online ARIMA algorithms, each of which employs an online convex optimization solver [11] in order to iteratively learn suitable model parameters γ. We refer to γ^t the current coefficients for a given time step t. The first algorithm employs the Online Newton Step (ONS) procedure [12], while the second algorithm utilizes Online Gradient Descent (ODG) [13]. While the ODG variant is computationally more efficient, the ONS approach provides a more favourable regret boundary that entails more accurate predictions. Moreover, choosing a suitable learning rate is harder in the gradient descent setting, yet crucial to achieve a good model approximation. Nevertheless, the ODG approach might be favourable when the lag-window is large.

IV. Unsupervised Anomaly Detection

For anomaly detection, we assume that most data within the monitored system are normal. Thus, we like to learn the normal behaviour of the system and recommend drastic changes. Let function $s : X \to X$ represent perfectly the given data stream, such that for a given data point $x \in X$ the prediction is always $s(x) = x$. In order to approximate the latest data stream, we use the Online ARIMA model to be able to predict the current data point based on the historic data.

We specify our ARIMA model via two parameters, namely the window size w, as well as the differencing depth d. The original ARIMA model utilizes three parameters, k, d and q, to designate past lags and errors (k and q respectively) being considered and differencing depth. We designate $k+m$ as our window size w, as it specifies the amount of relevant lagged data points. The parameters of window size, as well as differencing depth are statically defined when initializing the model.

In each time step a new data point X_t is observed, for which the model is then adapted. In order to train the model, the linear factors γ_i are adjusted according to the ONS procedure given by Hazan et al. [12]. We compare the received data point X_t to its forecasted value $\tilde{X}_t$ using the squared distance as loss metric. The loss gradient according to γ is then computed and utilized to execute the Newton Step[1], where the Matrix A corresponds to the Hessian Matrix H. We recognize the domain space for our coefficients as continuous, and thus omit the projection proposed in the original algorithm. In contrast to Liu et al., we denote no discrete feature space for the coefficients and assume them to be continuous. We therefore randomly initialize the coefficients ($\gamma_i \in [-0.5, 0.5]$, $\forall i \in [1,\ldots,w]$) and update them ensuing each observation.

Upon receiving the true value of the data point X_t, we update the model based on the prediction error (loss). In accordance with the algorithm described by Liu et al. we calculate:

$$\gamma^{t+1} = \gamma^t - \frac{1}{\eta} \cdot A_t^{-1} * \nabla t \ , \tag{4}$$

where $A_t = A_{t-1} + \nabla t * \nabla t^T$, and $\nabla t := \nabla l_t[\gamma^t](X_t, \tilde{X}_t)$. As the inversion of the pseudo-Hessian Matrix A is computationally expensive, we utilizing the Shermann Morrison formula [14] (eq. (5)) leads to a more efficient computation of the Inverse function, thus alleviating the computational cost of the ONS approach.

$$(A + u * v^T)^{-1} = A^{-1} - \frac{A^{-1} * u * v^T * A^{-1}}{1 + v^T * A^{-1} * u} \tag{5}$$

We can thus store the matrix A_{t-1}^{-1} instead of A_{t-1} and compute the next A_t^{-1} via:

$$A_t^{-1} = (A_{t-1} + \nabla t * \nabla t^T)^{-1} = A_{t-1}^{-1} - \frac{A_{t-1}^{-1} * \nabla t * \nabla t^T * A_{t-1}^{-1}}{1 + \nabla t^T * A_{t-1}^{-1} * \nabla t} \tag{6}$$

Given the input vector $X_t = (x_t^1 \ x_t^2 \ldots \ x_t^n)^T$ or rather, a time step t, our model fundamentally needs to forecast $\tilde{X}_t = (\widetilde{x_t^1} \ \widetilde{x_t^2} \ldots \ \widetilde{x_t^n})^T$. In order to do so, we create an individual forecast for each datum x_t^i ($i \in [1,\ldots,n]$) respectively. Thus, we require an independent model for each metric, representing each individual dimension within a data point. Effectively, our initial multi-dimensional time series is split into m standalone single-dimensional time series. After publishing n discrete forecasts, we need to recompose them to form a joint prediction $\tilde{X}_t$.

Each individual model (Model 1, ..., Model n) implements the ARIMA equations for a single-dimensional data point x_t. Our forecast $\tilde{X}_t$ is given by:

$$\tilde{X}_t = \begin{pmatrix} \tilde{x}_t^1 \\ \tilde{x}_t^2 \\ \vdots \\ \tilde{x}_t^n \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{w} \gamma_i^t \nabla^d x_{t-i}^1 + \sum_{i=0}^{d-1} \nabla^i x_{t-1}^1 \\ \sum_{i=1}^{w} \gamma_i^t \nabla^d x_{t-i}^2 + \sum_{i=0}^{d-1} \nabla^i x_{t-1}^2 \\ \vdots \\ \sum_{i=1}^{w} \gamma_i^t \nabla^d x_{t-i}^n + \sum_{i=0}^{d-1} \nabla^i x_{t-1}^n \end{pmatrix} \tag{7}$$

where $\nabla x_i = x_i - x_{i-1}$ and let x_t^i denote the datum corresponding to metric i ($i \in [1,\ldots,n]$) in time step t.

For each discrete model ensuing schema holds: Upon receiving the ground trough data value x_t we incur the prediction loss for our forecast $\tilde{x_t}$. We define our loss $l_t^j(x_t^j, \tilde{x_t}^j)$ (loss corresponding to model j in time step t):

$$l_t^j(x_t^j, \tilde{x_t}^j) = (x_t^j - \tilde{x_t}^j)^2 = (x_t^j - (\sum_{i=1}^{w} \gamma_i^t \nabla^d x_{t-i}^j + \sum_{i=0}^{d-1} \nabla^i x_{t-1}^j))^2 \tag{8}$$

Given the loss, we update our model according to the previously discussed methods.Thus, the loss gradient is for updating the model's coefficients:

$$\nabla l_t^j[\gamma^t](x_t^j, \tilde{x_t}^j) = \begin{pmatrix} -2 \cdot (x_t^j - \tilde{x_t}^j) \cdot \nabla^d x_{t-1}^j \\ -2 \cdot (x_t^j - \tilde{x_t}^j) \cdot \nabla^d x_{t-2}^j \\ \vdots \\ -2 \cdot (x_t^j - \tilde{x_t}^j) \cdot \nabla^d x_{t-w}^j \end{pmatrix}$$

The approach uses a sliding window to accommodate slices of new data, while phasing out the older data thus maintaining a constant size of the data structure. The window stores the latest w data values $x_{t-w}, \ldots x_{t-1}$, as well as all the necessary differenced values $\nabla^i x_{t-j}$ ($\forall i \in [1,\ldots,d]$), $\forall j \in [1,\ldots,w]$). Effectively, only the last row ($\nabla^d x_{t-i}, \forall i \in [1,\ldots,w]$) and last column ($x_{t-1}, \nabla x_{t-1}, \ldots, \nabla^{d-1} x_{t-1}$) are necessary to perform a forecast. Our implementation also stores the residual values for the sake of simplicity. However, this could be omitted in order to optimize storage consumption.

As the model progresses through time, the historical data columns drop out of the window tail, while the future data will be added to the head of the window. For an observed data point x_t, its differenced values are computed prior to being integrated into the window. The computation can be performed in an iterative manner, following the formula $\nabla^i x_t = \nabla^{i-1} x_t \nabla^{i-1} x_{t-1}, \ i > 0$.

In order to differentiate anomalies, we calculate a reconstruction error based on the prediction and the measured value using the Euclidean distance. Large errors indicate a shift in the state, which are captured by the learned forecasting model. Thus, we compute a threshold, which indicates whether the error is large enough to be stated as anomaly. The threshold is based on the mean μ and standard deviation σ of the latest w errors. Thus, when the current error is larger than $\mu + s \cdot \sigma$, the data point is stated as anomaly, where s is a user defined

[1] Customary Newton Step: $\gamma^{t+1} = \gamma^t - \eta \cdot H[l_t(\gamma^t)]^{-1} * \nabla l_t(\gamma^t)$

sensitivity factor. Additionally, we provide the possibility for smoothing the results using a window of size v. Based on the latest results in the window, the most available entry will be recommended.

V. Evaluation Setup

For evaluating the applicability of different configurations of our approach, we built a testbed running several different components, which are available as open source solutions. The components are described next in detail.

A. Infrastructure and VNF Service

We use as cloud infrastructure a balanced and replicated Openstack[2] installation consisting of seven compute nodes and three controller nodes. The infrastructure consists of ten physical hosts possessing an Intel Xeon X3450 (4 cores, 2.66GHz), 16GB RAM, 3x 1TB disk, 2x 1GBit Ethernet connection. Through Openstack, it is possible to create virtual machines, in which we deploy our services. The virtual machines (Ubuntu 14.04) use 2 vCPU cores, 2GB memory, 20GB disk.

We deployed the open source implementation of an IP multimedia subsystem (IMS), called Clearwater[3]. Clearwater is considered as one of the first examples for virtual network functions, which makes it highly interesting to telco-providers. IMS is an emerging architecture for IP-based telecommunication services, such as voice-, video calls or messaging. The core implementation of Clearwater allows users to register, which can then initiate calls within the system. There exists the services named Bono, Sprout, Homestead and Ellis, which are considered within the experiments.

Bono functions as edge proxy, handling clients connections. Those can register via the SIP protocol in order to initiate calls. The requests are then routed to the Sprout service.

Sprout manages the different communications to the other internal services, e.g. requesting for authentication.

Homestead contains the client profile informations, which are needed to authenticate clients.

Ellis obtains the information for the management unit, such it functions as an account management system.

In order to simulate the usage of the IMS, we change every minute the number of client registrations and call initiations randomly. Furthermore, we deployed a replicated version, which is load balanced. Thus, the key services Bono and Sprout are deployed three times each, while single deployments are used for Homestead and Ellis. For each service installation, we created its own virtual machine. For monitoring, we collect inside each virtual machine the resource usage data parsing the proc filesystem, provided by the operating system. The data collection interval is 500ms.

B. Anomaly Injection

In order to evaluate the proposed approaches, we simulate a larger set of anomalous behaviour in the system. Thus, we developed an injection agent handling several different anomalies. The agent is placed inside each virtual machine and can execute the following anomalies:

Disk pollution (DP), temporary disk pollution (DP tmp) and HDD stress (HDD-S): Both anomalies are writing data into a log-file. For the temporary case, the files are deleted, while in the other case the file is still available. HDD stress describes an excessive hard disk usage by writing a file.

CPU stress (CPU-S), leak (CPU-L) and fluctuation (CPU-F): CPU stress immediately consumes excessively much CPU. CPU leakage describes an anomaly increasing in its intensity over time by consuming more and more CPU. The fluctuation of CPU constantly increases and decreases the CPU usage in order to provide a fluctuation behaviour.

Memory stress (M-S), leak (M-L) and fluctuation (M-F): Like the CPU, memory anomalies are also modelled with the same behaviours: immediately consuming high amounts of memory, growing consumption over time and fluctuating allocation of memory.

Fork flooding leak (FFL) and fluctuation (FFF): A process starts to create child processes over time. For the leakage variant, those child processes create more processes, while in the fluctuation variant children are also removed partly.

Large file download (DL): A process is started downloading a large file, resulting in high network traffic.

File pointer wasting (FPW): By requesting file pointers without releasing them, creates a leakage of the pointer IDs over time.

Normal behaviour (NR): After introducing all the different anomalies, there will be always phases between those, where the whole system runs without any anomaly, which we consider as normal behaviour.

Through using several simulated anomalies with different behaviours, we like to show the robustness of our approaches quality. Notably, the intensity increase of each anomaly can be configured and is randomly selected for the current evaluation.

C. Analysis Pipeline Configuration

As described above, we collect metrics from the proc filesystem, which consists of a large number of values. Before performing the anomaly detection on the data, we do the following preprocessing steps:

Feature selection: We selected the following metrics for further analysis: CPU percentage, disk-io, disk-io time, load from the previous 1s and 5s, memory usage and percentage, network-io bytes, packets, number of errors, number of dropped packets.

Feature creation: In order to gain more knowledge from the given metrics, we also created new metrics based on the collected ones. The slope of the last 60s of memory providing information about the change of memory over time. Furthermore, we added metrics representing information about the packet size and relations between CPU usage, network traffic and disk usage.

Normalization: Min-max scaling is performed for each metric, resulting in a scaled metric $\hat{x} = \dfrac{x - x_{min}}{x_{max} - x_{min}}$ for an

[2] https://www.openstack.org/

[3] http://www.projectclearwater.org/

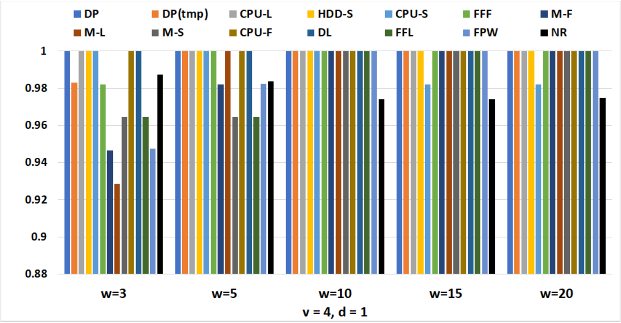

Fig. 1. Detection accuracy and normal rate (NR), $v = 4$, $d = 1$

incoming data point x, where x_{min} and x_{max} represent the minimum and maximum values for the individual metrics. These can be inferred over time or set by an expert.

Based on the evaluation setup, we collected data of all services for 72h. The first 20 minutes of data consists of normal data grace time. Afterwards, the anomalies were injected in a round-robin manner to the individual service hosts for 3 minutes. Between the execution of anomalies, a cool down of 1 minute is performed. For evaluation purposes, we used datasets for each individual host, containing anomalies running on that specific host and time frames where the system runs in normal mode.

Parameters: Next, we show the chosen hyper-parameters on which we perform a grid search for the best configuration.

Window size: $w \in 3,5,10,15,20$. We refrain from evaluating sizes larger than 20 as we like to learn and predict in real-time expect to performance.

Differencing depth: $d \in 0,1,2,3$. We limit our differencing depth to a ceiling of 3 as we do not expect increased differencing to be beneficial for single-dimensional data.

Smoothing values: $v \in 2,4,6,8$. We expect for larger smoothing ranges poor detection rates, which can be explained by the fact that ARIMA models adapt their data view quickly thus making it unlikely that many successive anomaly predictions (which are induced by large deviations between forecast and true observation) occur.

Threshold sensitivity: $s = 0.8$. A fine grained calibration of s could potentially optimize joined performance over detection and normal rate. However, this entails considerable tuning efforts which we leave to further research as our main focus is on the influence of the ARIMA model parameters $(w,\ d)$.

VI. Evaluation Results

We show three setups with fixed smoothing and differencing depth, where we iterate across all considered window sizes. The first two examples utilize smoothing value $v = 4$ and show a comparison of the depths $d = 1$ and $d = 2$. Lastly, an overview for a large smoothing value $v = 8$ with $d = 3$ is presented. All setups are accompanied by the average detection time (ms) of each anomaly type as well as the average false alarm duration $(10^{-1} \cdot \mathrm{s})$.

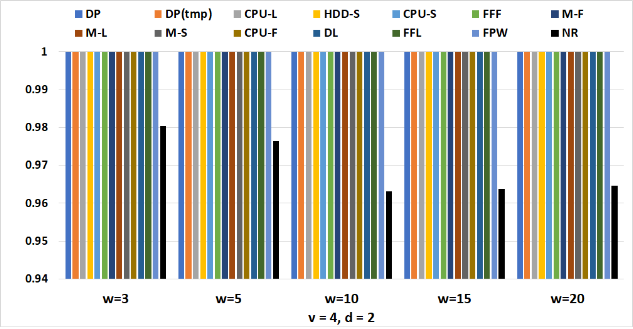

Fig. 2. Detection accuracy and normal rate (NR), $v = 4$, $d = 2$

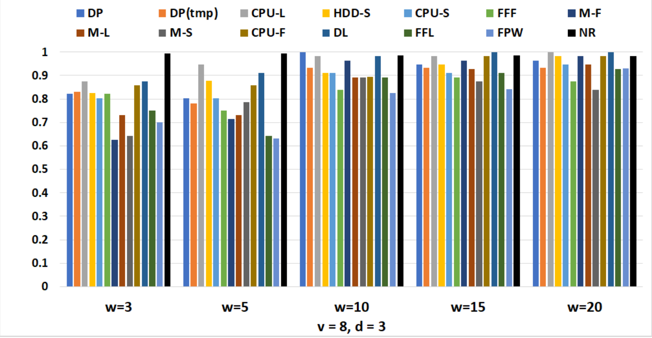

Fig. 3. Detection accuracy and normal rate (NR), $v = 8$, $d = 3$

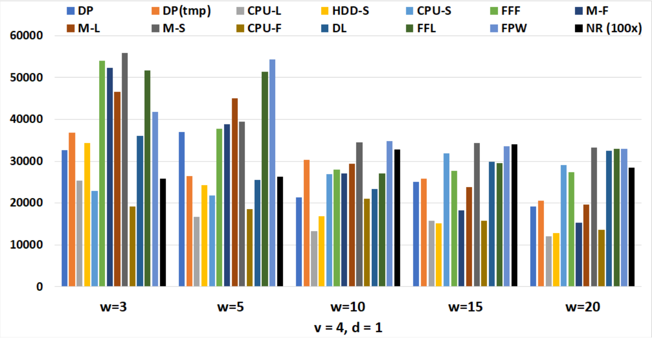

Fig. 4. Detection time after anomaly started in [ms] and false alarm duration (NR 100x), $v = 4$, $d = 1$

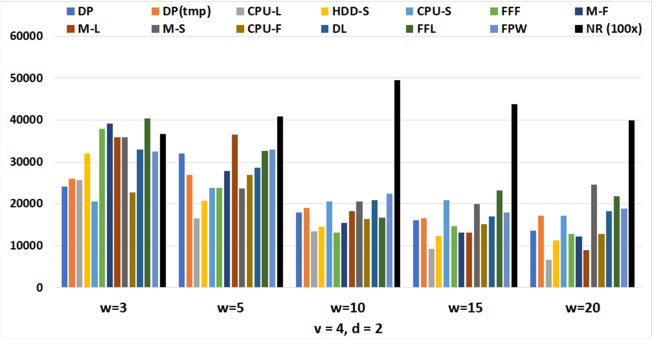

Fig. 5. Detection time after anomaly started in [ms] and false alarm duration (NR 100x), $v = 4$, $d = 2$

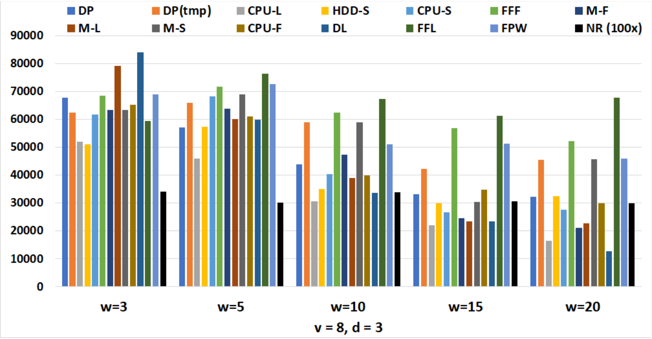

Fig. 6. Detection time after anomaly started in [ms] and false alarm duration (NR 100x), $v = 8$, $d = 3$

Figure 1 shows the detection rates for the parameters $v = 4$ and $d = 1$. It is noticeable that very small window sizes (< 10) perform considerably worse than the examined larger window ranges (10 and above). Window size 10 provided the best accuracy with windows $w = 15$ and $w = 20$ performing equally well. On the flip side, the normal rates decrease as detection accuracy increased, suggesting a trade-off on sensitivity. From the average time to detection depicted in Figure 4, we can deduce that larger window sizes are not only more accurate but increasingly reactive. Considering both detection accuracy and reaction time we surmise that $w = 20$ performs most efficiently for depth $d = 1$.

In our second example setup (Fig. 2 and Fig. 5), we assume the smoothing range $v = 4$ as well, however we increase our differencing depth to $d = 2$. This drastically increases detection accuracy across all window sizes (accuracy 100%). For $w \geq 10$ the normal rates suffer considerably, wherefore in this setup we favour the use of a smaller window size. The detection time is reduced as well, which can be attributed to the higher observed detection sensitivity. While the larger windows react faster, they impose higher false alarm durations. As normal rates were also low, this suggests that larger window sizes were more sensitive for $d = 2$ as well, yet perhaps oversensitive. We favour $w = 3$ or $w = 5$ that offer the best trade-off between detection accuracy and normal rate.

Finally, in Figures 3 and 6 we show the results for a setup with a large smoothing range $v = 8$ and differencing order $d = 3$. The purpose of increased smoothing is to elevate the normal rate, which was comparably low in the previous examples. Over the course of our evaluations, we could observe that higher differencing depths incited more sensitivity and thus detection accuracy (see Fig. 4 vs. Fig. 5), at the cost of an increased false alarm count. Fig. 3 shows that the normal rate is indeed very high (99.8%), yet the detection accuracy suffers heavily (note the scale difference, accuracy dropping as low as 60%). As such, the average detection time naturally increases due to a lowered sensitivity (see Fig. 6). Smoothing overly dominates the model behaviour, making large ranges unsuitable for detection. Smoothing ranges that are too small ($v = 2$) however allow for models that are too oversensitive. While they convincingly depict 100% detection accuracy, they also incur an abundance of false alarms (normal rate $91-96\%$), rendering them unfavourable compared to the discussed setups with $v = 4$ (Fig. 4 and 5).

To summarize, we recommend to use a configuration $v = 4$, $d = 2$ and $w = 3$, as those show an optimal detection accuracy (100%) and low number of false alarms (<2%).

VII. Conclusion

This work presented an unsupervised anomaly event detection approach for service monitoring. We showed how the Online Arima forecasting algorithm can be used for anomaly detection. This approach was evaluated using a virtual network function use case applying several different anomalies into the service infrastructure. The results show, that the approach achieves high rate to detect anomalies while giving a small number of false alarms.

In future, we like to evaluate this approach on further NFV services and extend the evaluation on the private cloud provider hosts in order to monitor the physical machines. Furthermore, we like to investigate in more detail seasonal behaviours of load usages as they appear often in real-world scenarios. In future, we hope that such automatic methods should provide the possibility for zero touch administration.

References

[1] E. Bauer, X. Zhang, and D. A. Kimber, *Practical system reliability*. John Wiley & Sons, 2009.

[2] M. H. Ghahramani, M. Zhou, and C. T. Hon, “Toward cloud computing qos architecture: Analysis of cloud systems and cloud services,” *IEEE/CAA Journal of Automatica Sinica*, vol. 4, no. 1, pp. 6–18, 2017.

[3] A. Gulenko, M. Wallschläger, F. Schmidt, O. Kao, and F. Liu, “A system architecture for real-time anomaly detection in large-scale nfv systems,” *Procedia Computer Science*, vol. 94, pp. 491–496, 2016.

[4] C. Liu, S. C. H. Hoi, P. Zhao, and J. Sun, “Online arima algorithms for time series prediction,” in *Proceedings of AAAI Conference on Artificial Intelligence*, ser. AAAI’16. AAAI Press, 2016, pp. 1867–1873.

[5] C. Sauvanaud, K. Lazri, M. Kaâniche, and K. Kanoun, “Anomaly detection and root cause localization in virtual network functions,” in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 2016, pp. 196–206.

[6] J. Liu, S. Chen, Z. Zhou, and T. Wu, “An anomaly detection algorithm of cloud platform based on self-organizing maps,” *Mathematical Problems in Engineering*, vol. 2016, 2016.

[7] 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*. ACM, 2012, pp. 191–200.

[8] D. Cotroneo, R. Natella, and S. Rosiello, “A fault correlation approach to detect performance anomalies in virtual network function chains,” in *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*. IEEE, 2017, pp. 90–100.

[9] J. D. Hamilton, *Time series analysis*. Princeton University Press, 1994, vol. 2.

[10] E. J. Hannan, *Multiple time series*. John Wiley & Sons, 2009, vol. 38.

[11] S. Bubeck, “Introduction to online optimization,” *Lecture Notes*, pp. 1–86, 2011.

[12] E. Hazan, A. Agarwal, and S. Kale, “Logarithmic regret algorithms for online convex optimization,” *Machine Learning*, vol. 69, no. 2-3, pp. 169–192, 2007.

[13] M. Zinkevich, “Online convex programming and generalized infinitesimal gradient ascent,” in *Proceedings of International Conference on Machine Learning (ICML-03)*, 2003, pp. 928–936.

[14] J. Sherman and W. J. Morrison, “Adjustment of an inverse matrix corresponding to a change in one element of a given matrix,” *The Annals of Mathematical Statistics*, vol. 21, no. 1, pp. 124–127, 1950.