Anomaly Detection and Classification using Distributed Tracing and Deep Learning

Sasho Nedelkoski*, Jorge Cardoso†, Odej Kao*
*Complex and Distributed IT-Systems Group, TU Berlin, Berlin, Germany
Email: {nedelkoski, odej.kao}@tu-berlin.de
†Huawei Munich Research Center, Huawei Technologies, Munich, Germany
Email: jorge.cardoso@huawei.com

***Abstract*—Artificial Intelligence for IT Operations (AIOps) combines big data and machine learning to replace a broad range of IT Operations tasks including availability, performance, and monitoring of services. By exploiting log, tracing, metric, and network data, AIOps enable detection of faults and issues of services. The focus of this work is on detecting anomalies based on distributed tracing records that contain detailed information for the availability and the response time of the services. In large-scale distributed systems, where a service is deployed on heterogeneous hardware and has multiple scenarios of normal operation, it becomes challenging to detect such anomalous cases. We address the problem by proposing unsupervised, response time anomaly detection based on deep learning data modeling techniques; unsupervised dynamic error threshold approach; tolerance module for false positive reduction; and descriptive classification of the anomalies. The evaluation shows that the approach achieves high accuracy and solid performance in both, experimental testbed and large-scale production cloud.**

***Index Terms*—AIOps; anomaly detection; service reliability; time series; distributed tracing; autoencoders; RNNs; GRUs; CNNs.**

I. Introduction

The increasing number of IoT applications with dynamically linked devices and their embedding in real-world (smart) environments drive the creation of large multi-layered systems. As a consequence, the complexity of the systems is steadily growing up to a level, where it is impossible for human operators to oversee and holistically manage the systems without additional support and automation. Uninterrupted services with guaranteed latency, response times, and other QoS parameters, are however mandatory prerequisite for many of the data-driven and autonomous applications. Therefore, losing control is not a feasible option for any system or infrastructure.

The large service providers are aware of the need for always-on, dependable services and they already deployed numberless measures by introducing additional intelligence to the IT-ecosystem. For example, by employing network reliability engineers (NRE), site reliability engineers (SRE), by using automated tools for infrastructure monitoring, and developing tools based on artificial intelligence (AIOps) for load balancing, capacity planning, resource utilization, storage management, and anomaly detection.

The next piece in the puzzle aims at rapidly decreasing the reaction time in case an urgent activity of a system administrator is necessary. That usually involves performance problems, component/system failures (e.g., outages, degraded performance), or security incidents. All these examples describe situations, where the system operates outside of the normal, expected or pre-defined behaviour. Thus, the system exposes that an anomaly must be detected and recognized, before it leads to a service or a system failure.

The foundation for AIOps systems is the availability of suitable and descriptive data, which is typically observed by three core components: tracing, logging, and resource monitoring metrics. The tracing component produces events (spans) containing information on the execution path and response time. The logging data represents interactions between data, files, or applications and is used to analyze specific trends or to record events/actions for a later forensic. The resource monitoring data reflects the current utilization and status of the infrastructure, typically as cross-layer information regarding CPU, memory, disk, and network throughput and latency. Most of the current AIOps platforms apply deep learning solely on monitoring data [1], [2], as this data is simple to collect and interpret, but not sufficient for a holistic approach. We aim at exploring an additional path for anomaly detection using a second category of data, namely the tracing data collected during the execution of system operations. Tracing technologies [3]–[5] generate events to externalize the state of the system by combining performance data from the end-to-end execution path with structured and causally related execution traces. We are confident that such data can improve the anomaly detection, root-cause analysis, and remediation in the system. It contains detailed information for individual services and the causal relationship to other related services that form part of the trace.

The focus of the study is to tackle the problem of anomaly detection in real-world tracing data. It faces several challenges, including the lack of labeled data, concept drift, and concept evolution. Other major sources of difficulties emerge due to the low signal-to-noise ratio, the presence of multiple frequencies and multiple distributions, the large number of distinct time series generated by microservice applications, and the presence of concept drifts. The signal-to-noise ratio is typically very low as many different components affect the response time of microservices such as switches, routers, memory capacity, CPU performance, programming languages, thread and process concurrency, bugs, and volume of user

requests. Multiple frequencies are correlated with system and user behaviour since request patterns are different, e.g., from hour to hour due to business operation hours, from day to day due to system maintenance tasks, and from month to month due to cyclic patterns. For these reasons, the utilization of unsupervised approaches is required. In such scenarios where anomaly detection is being used as diagnostic tool, a degree of additional description is required. Identifying the potential anomaly in the service is of limited value for the operators without having more detailed explanation.

Contributions. We provide an algorithm that adapts and extends deep learning methods from various domains. This work focuses on anomaly detection from tracing data in large-scale distributed systems, but can also be used in other applications involving anomaly detection on time series data containing multiple normal operating scenarios. We show the capability of the Auto-Encoding Variational Bayes (variational autoencoder, AEVB) to learn multiple complex distributions representing normal behavior over longer period of time and detect anomalies by employing a dynamic, probability-based, error threshold setting. Furthermore, we propose combination of the threshold setting and post-processing that aims to reduce the number of false positives. Lastly, we present a classification module and provides descriptions for the detected anomalies.

The remaining of the paper is structured as follows. In section II, we provide the related work for the field of anomaly detection. In sections III and IV, we present the preliminary knowledge and our proposed methodology. Section V summarizes the performance evaluation in terms of speed and accuracy for different types of anomalies on both experimental and real-world production cloud data.

II. Related work

Tracing technologies for distributed services record information about all the individual components participating on an e.g., user request (initiator) within the system. Two classes of solutions have been proposed to aggregate these information so that one can associate all record entries with a given initiator, black-box and annotation-based monitoring schemes [3]. Black-box schemes [6]–[8] assume there are no additional information other than the message record described above, and use statistical regression techniques to infer that association. Annotation-based schemes [5], [8]–[10] rely on applications or middleware to explicitly tag every record with a global identifier that links these message records back to the originated request. We use the annotation based system (Zipkin based on Dapper [3]), which relies on proper service instrumentation.

While the anomaly detection on other categories of data like log and metric are part of previous research [1], [2], [11]–[15], the related work on time series and the structural anomaly detection in trace data is still limited.

Anomaly detection for services have been studied exhaustively during many years on different kinds of data. In general, we distinguish between statistical and machine learning methods. The machine learning approaches can be divided into two general categories [1], supervised [16]–[19] and unsupervised [20]–[23].

Vallis et al. [24] proposed a novel approach, which builds on Extreme Studentized Deviate test (ESD), for detecting anomalies in long-term time series data. The approach requires the detection of the trend component. This technique is similar to most of the statistical methods, which have limitations when they are applied to large systems based on service-oriented and microservice architectures. These systems produce time series data with high noise and with more than a single normal behavior in the signal. Specifically, if the time series has more than two different normal (expected) scenarios of operation, the algorithm would not be able to capture this information.

Supervised methods use labeled data to train machine learning models. The anomaly detection algorithms are classification models trained by data containing the information whether the data point is an anomaly or not. For practical usage, the labelling by experts or injection of anomalies either is not sufficient (evolving time series, concept drifts) or may harm the running system. Therefore, unsupervised methods are investigated, having the positive properties of performing the same task, but not using labeled input data.

Recently, deep learning techniques are increasingly investigated because of their success in range of domains. In that direction, Malhotra et al. [25] used stacked recurrent hidden layers to enable learning of higher level temporal features. They presented a model of stacked Long Short-Term Memory (LSTM) networks for anomaly detection in time series. A network was trained on non-anomalous data and used as a predictor over a number of time steps. The resulting prediction errors were modeled as a multivariate Gaussian distribution, which was used to assess the likelihood of anomalous behavior. The efficacy of this approach was also demonstrated on four datasets.

Xu et al. [26] show the usability of variational autoencoders for anomaly detection and triggering of timely troubleshooting problems on Key Performance Indicator (KPI) data of Web applications (e.g., page views, number of online users, and number of orders). They proposed Donut, an unsupervised anomaly detection algorithm based on AEVB. Furthermore, Hundman et al. [27] show the use of LSTM recurrent neural networks for spacecraft anomalies on multi-variate telemetry data.

Existing supervised and unsupervised auto-regressive approaches fail with data of small signal-to-noise ratio and autocorrelation either by learning only the running mean or by not preserving the order in time series. In similar direction as of Xu et al. and Hundman et al., we combine the methods from both and show that the integration of Gated Recurrent Units (GRUs, simplified LSTMs) with variational autoencoder produce results which are able to meet the accuracy and performance requirements. Of course, the inclusion of preprocessing and postprocessing improves the accuracy by reducing the amount of false positives.

III. Preliminaries

Systems based on microservices or service-oriented architectures consists of several services connected by a network, providing a larger system application. To monitor the user requests with a detailed description of different participating microservices, distributed tracing technologies are utilized.

A trace $\mathrm{T} = (\mathrm{E}_0, \mathrm{E}_1, \ldots, \mathrm{E}_\mathrm{i})$ is represented by an enumerated collection of events. Each event is represented by key-value pairs (k_i, v_i) describing the state, performance, and other characteristics of a service at a given time t_i. The events contain contain a *timestamp* when the particular service was invoked, a *response time*, and a *http URL* among the other meta information (e.g., *host IP, service name etc.*). Depending on the request, traces can have different lengths and services invoked. The *response time* is one of the most important attributes of the event, e.g., if its value which characterizes the intra-service calls in the system suddenly increases at times t_i, t_{i+1}, t_{i+2}, it may indicate a problem with the underlying distributed system.

Let us assume that we observe two traces: $\mathrm{T}_1 = (\mathrm{E}_{\mathrm{url-a_1}}, \mathrm{E}_{\mathrm{url-b_1}}, \mathrm{E}_{\mathrm{url-c_1}})$ and $\mathrm{T}_2 = (\mathrm{E}_{\mathrm{url-a_2}}, \mathrm{E}_{\mathrm{url-b_2}}, \mathrm{E}_{\mathrm{url-c_2}})$. Each *http URL* recorded in the platform is a source of events. Events of the same type are clustered together by their *http URL* to form a time series $\mathrm{TS}_1 = (\mathrm{E}_{\mathrm{a_1}}, \mathrm{E}_{\mathrm{a_2}})$.

Further, let us assume that we record the following two *http URLs* from two events:

- 1.1.1.5/tag_1/group_id1/tag_2/service1
- 1.1.1.6/tag_5/group_id2/tag_4/service1

We use regular expression for each of the events having *http URL* in form of {host IP}/{tag_id}/{group_id}/{tag_2}/service1 and assign them to the same cluster The same procedure is done with other events such as those that belong to {host}/groups/{group_id}/logs. We name such groups as endpoints and to each we assign *cluster IDs* represented by the regular expression. The time series formed by the groups of events, having the properties as explained in Section I, are used to study the dynamics of the system and to detect anomalies.

Anomaly detection on time series consisting of the service's *response time* can be formulated as follows: For any time t given historical observations $x_t = \{e_{t-w}, e_{t-w+1}, ..., e_t\}$, where w is the sliding *window size* and e_t is the event's *response time* at time t, determine whether an anomaly occurs or not (1/0). We use a sliding window to break the time series into fixed-size inputs, required for the autoencoder. The sequential order of the points inside the window is important. Therefore, we combine the AEVB with the ability of the RNNs for extracting temporal information from sequential data. An anomaly detection algorithm typically computes a real-valued score indicating the certainty of having anomaly, e.g., $p(anomaly = 1|e_{tW}, ..., e_t)$, instead of directly computing, whether the window represents an anomaly.

A. Variational autoencoder for anomaly detection

An autoencoder is an unsupervised neural network architecture. It applies backpropagation like the standard feed forward neural network, setting the output (target) value to be equal to the inputs i.e. $y_t = x_t$ [28]. The identity function seems a trivial function to be learn, but by placing some constraints, such as limiting the number of hidden units, or putting regularization, interesting features from the data can be extracted. A typical architecture of an autoencoder is shown in Figure 1, where h is called latent representation or bottleneck of the autoencoder.

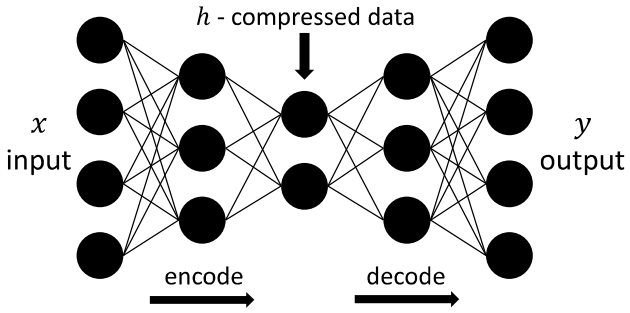


Fig. 1. Architecture of an autoencoder network. The x and y can be of any type, in this paper $x = y$ is time series data.

By training on non-anomalous data, the autoencoder learns and is able to produce good reconstructions on new non-anomalous samples (low error). If we use the same model to predict the reconstruction for an anomalous sample, then the reconstruction error will be larger.

A variational autoencoder (AEVB) [29] is a deep neural network architecture that can learn complex representations from data without supervision. AEVB is composed of an encoder and decoder, both are neural networks, and contain a loss function. Instead of mapping the input vector onto a fixed vector as in the usual autoencoders, the model maps any input into a predefined distribution. Moreover, the bottleneck vector in the variational autoencoder is replaced by two vectors of the same size. One of them representing the mean and the other representing the variance of the distribution. So, whenever we need the output of the encoder in order to feed into the decoder network, we need a sample from the distribution, defined by the mean and standard deviation vectors that represent the latent low-dimensional space. Let us assume that we have a dataset X of samples from a distribution parametrized by a ground truth generative factor. The variational autoencoder aims to learn the marginal likelihood of the data in a generative process:

$$\underset{\phi,\theta}{\text{maximize}} \quad \mathbb{E}_{q_\phi(z|x)}\left[\log_{p_\theta}(x|z)\right] \tag{1}$$

Where ϕ and θ parametrize the distributions of the VAE encoder and the decoder respectively. Furthermore, the complete loss function is given by:

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}) = \mathbb{E}_{q_\phi(z|x)}\left[\log_{p_\theta}(x|z)\right] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})) \tag{2}$$

The loss function, as written in (2), consists of two terms. The first term represents the reconstruction loss, which is part

of any autoencoder architecture, except we have the expectation operator, because we are sampling from the distribution. The second term is the Kullback – Leibler divergence that ensures close mapping to a predefined distribution.

Recently, there is an increasing adoption of unsupervised, generative machine learning models for anomaly detection. Similarly, the variational autoencoder (AEVB) first learns the normal scenario (one, or many) [26]. Then, conditioned on its input is able to generate reconstructions. By setting a threshold on the reconstruction error, we are able to classify a given window of *response time* as anomaly or normal.

B. Recurrent variational autoencoder

Recurrent neural networks (RNNs) [30] are a type of neural networks where the connections between neurons form a directed cycle. They are capable of learning features and long term dependencies from sequential and time-series data. A typical architecture of the RNN is shown in Figure 2. Each step in the unfolding is referred to as a time step, where x_t is the input at time step t. RNNs can take an arbitrary length sequence as input, by providing the RNN a feature representation of one element of the sequence at each time step. s_t is the hidden state at time step t and contains information extracted from all time steps up to t. The hidden state s is updated with information of the new input x_t after each time step: $s_t = f(Ux_t + Ws_{t1})$, where U and W are vectors of weights and f is the non-linear activation function. The most used RNN types in practice are RNNs with LSTM (Long Short-Term Memory) [31] or GRU (Gated Recurrent Unit) [32] cells, which we use in this paper as well.

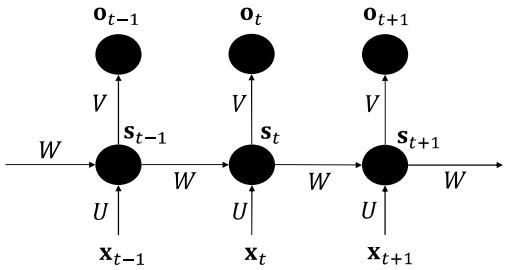


Fig. 2. Architecture of RNN.

Recurrent variational autoencoder [33] is combination of AEVB and RNN. The encoder is a recurrent neural network (RNN) that processes the input sequence $x_t = \{e_{t-w}, e_{t-w+1}, ..., e_t\}$ and produces a sequence of hidden states $\{h_{t-w}, h_{t-w+1}, ..., h_t\}$. The parameters of the distribution over the latent code is then set as a function of h_t. The decoder, uses the sampled latent vector z to set the initial state of a decoder RNN, which produces the output sequence $y = y_1, y_2, ..., y_t$. The model is trained both to reconstruct the input sequence and to learn an approximate posterior close to the prior like in a standard variational autoencoder.

IV. Response Time Anomaly Detection

The following methods form the core components of our unsupervised anomaly detection approach for microservice or service oriented systems observed by distributed tracing technology. First, the time series data is preprocessed and a neural network model is trained on it to capture the normal system behavior. Based on this model, the predictions for the reconstruction are obtained. Then, a probability based, adaptive threshold method is used to determine whether resulting prediction errors represent anomalies for individual services. Further, a post-processing strategy, incorporated in a tolerance module, is used to mitigate false positives. Lastly, we provide anomaly pattern classification to provide descriptive and useful analysis results. We divide the proposed methods in four core steps or modules, that exchange the results in-between.

- *time series preprocessing*
- *model training*
- *test-time prediction*
- *faulty pattern classification*

For simplicity, we will describe the methods through the lens of a single time series. Given K time series, the solution scales since is meant to be applied to every time series in parallel.

A. Time series preprocessing

This step involves two parts, preprocessing in *model training* and *test-time prediction*. The module queries the latest N data points (events) belonging to the same *cluster ID* (time series) and forwards it into a three stage pipeline: data cleaning, normalization and noise reduction.

Tracing events are JSON objects, but in dependence of the service instrumentation they might have a slightly different structure. Common for all are the *response time*, which is extracted for further processing. We assume that most of the time the services in the system are in normal mode of operation. That is true in real-world systems where failures happen rarely. However, the large amount of events in the time series and the fact that proper training of neural networks requires normalization, leads to obligation of having an outlier removal technique. The presence of a strong outlier, will lead to values clamped to zero after the normalization. Therefore, events having *response time* greater than three standard deviations from the mean are removed from the training batch. Next, we normalize the values by using min-max scaling (0, 1) to ensure that we stay in the positive range of values for the *response time*. In contrast, using standardization might produce negative values that do not have natural meaning when we deal with *response time* (no negative time). Normalization is required and makes the optimization function of the neural network well-conditioned, which is key for convergence [34]. Min-max normalization is given with the following equation:

$$X_{t,scaled} = \frac{X_t - min(X)}{max(X) - min(X)} \tag{3}$$

where $min(X)$ and $max(X)$ are saved and then used for the normalization in *test-time prediction*. Lastly in the pipeline,

we apply smoothing for noise removal and robustness to small deviations. The time series is convolved with Hamming smoothing filter defined with its optimal parameters [35] and size M as:

$$w(n) = 0.54 - 0.46 \cdot cos\left(\frac{2\pi n}{M-1}\right), 0 \leq n \leq M-1 \quad (4)$$

We use smoothing with size of the window $M = 12$, but one can adjust the size depending on the noise.

For *test-time prediction* the preprocessing is executed on each new recorded event. During test-time, the event follows the same preprocessing steps as for the *model training* except the normalization where $min(X)$ and $max(X)$ are the saved values during *model training* part.

Time series partitioning: After the steps in the preprocessing, we define *window size*, which represents number of points in a sliding window that needs to be considered for evaluation. The window with the predefined size and stride is applied to the time series. This results in training data shape of: $(N - window\ size, window\ size, 1)$. The data in such format is then feed into the neural network for training. In *test-time prediction*, each *window size* number of events are fed to the network for prediction.

B. Model architecture

The architecture of our proposed neural network is shown in Figure 3 and described in following.

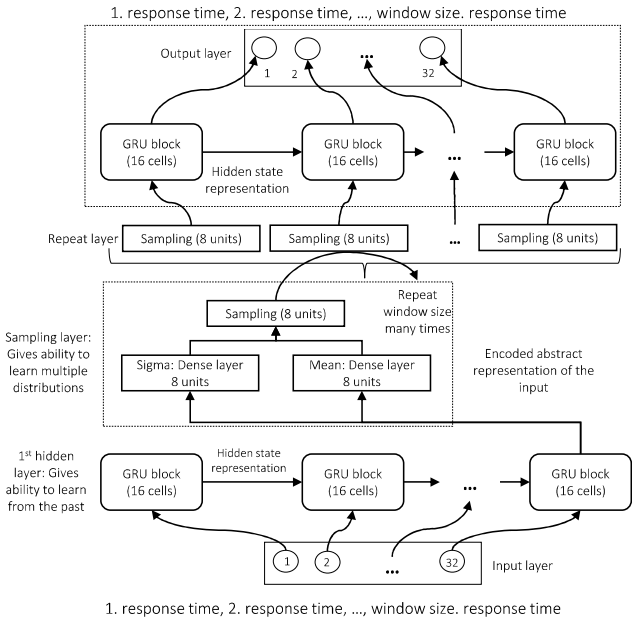


Fig. 3. Model architecture.

Input layer: has *window size* number of units, each containing the *response time* as input. In Figure 3, we use $window\ size = 32$.

First hidden GRU layer: contains $(window\ size/2)$ 16 GRU cells for each timestep in the input window. Each of the *window size* input units is fed to the corresponding GRU. In the first timestep $T = 0$, the 0^{th} *response time* is fed. The abstract representation learned in the 16 GRU cells is then propagated to the next timestep $T = 1$, where the 1^{st} *response time* of the window is fed and so on. Here, we have the ability to condition the reconstruction of the next point given the past points. In such way, that in the last timestep we have abstract representation of the window of points, which has salient information for that part of the time series.

Sampling layer: Represents the key part in order to be able to learn multiple distributions (model of models). This layer consists of $(window\ size/4)$ 8 units for the mean and for the variance. The sampling layer just performs sampling from multivariate normal distribution with the given mean and variance.

Repeat layer: Repeats the Sampling layer *window size* times, which is needed to be fed into the last hidden (GRU) layer.

Output/GRU layer: Here, the network takes the output from the previous layer as input, learns abstract representation and as output have the same *window size* number of input timesteps only with the *response time* as feature.

1) Training details: We observed that the required number of data points in particular time series used to produce good model in training should be more than 1000. The training data is split into two parts in sequential order. The smaller part or 20% goes for estimating parameters and tuning the model. We train the model for 1000 epochs and choose the one with the best validation score. The solution uses Adam optimizer with learning rate of 0.001, which are the standard values for training deep neural networks [28]. As mentioned, the error function which we optimize is described in Section III. As last step when the training is finished, the model is saved and used in *test-time prediction*.

2) Dynamic error threshold: The difference between a prediction and an observed parameter value vector is measured by the mean square error (MSE) which is given with the following equation.

$$MSE = \frac{1}{window\ size}\sum(x_i - y_i)^2 \quad (5)$$

Instead of setting a magic error threshold for anomaly detection purpose, we use the validation set for threshold setting. For each window/sample in the validation set, we apply the model produced by the training set and calculate the MSE between the prediction (reconstruction) and the actual sample. At every time step, the errors between the predicted vectors and the actual ones in the validation group are modeled as a Gaussian distribution. Assume that the validation data has 1000 windows of with $window\ size = 32$. The MSE for all of them will produce array of 1000 error values. Next, we estimate the mean and the variance for the MSE scores and save them on the disk along with the model. These values are used in *test-time prediction*. In *test-time prediction* if the error between a reconstructed and an observed window of events is within a high-level of confidence interval of the above

Gaussian distribution is considered as normal, otherwise as anomaly.

C. Test-time prediction

Previously, we already showed the architecture, model training and dynamic threshold setting. After having the trained model, this module receives data from the preprocessing module described previously. The latest model along with the saved training parameters are loaded, and used for prediction. For each new event, the past values forming a window $x_t = \{e_{t-w}, e_{t-w+1}, ..., e_t\}$ are fed as input for prediction. The reconstruction error MSE_{test} and the probability under the Gaussian are computed:

$$P_{test} = 1 - P(X > MSE_{test}) \tag{6}$$

Remembering that the parameters of the probability density function μ and σ^2 are computed as parameters in the training step.

1) Tolerance: false positive reduction: In large-scale system architectures, there are thousands of events recorded in short period of time and there are cases where it might happen that the *response time* is greater compared to the expected time. If it is only a single anomalous point in the time series or even few of them with increased service *response time*, does not mean that anything is wrong with the service. For example, that can be a small bottleneck in the disk usage or in one of the many components or services. Aiming to detect anomalies that have larger impact, enables the DevOps to pay only attention to the most critical potential failures.

We define the *tolerance* and *probability error threshold* as parameters. The *tolerance* represents the allowed number of anomalous windows that have $P(x)$ greater than the *probability error threshold* before it flags the whole period as anomalous. In practical scenarios, the *tolerance* parameter usually ranges from 1 to 100, but it is dependent on the dynamics of the system. The probability outputs P_{test} are kept in queue with the same size (*tolerance*) for each new window. Each time, a new sample is shown to the network to be reconstructed, assigned with the probability of being anomalous and is added to the queue, the tolerance module checks whether the average probability:

$$P_m = \frac{1}{tolerance} \sum_{i}^{tolerance} P_{test(i)} \tag{7}$$

of all the points in the queue is greater than the error threshold. If this is the case, the submodule flags this part of the time series as unstable and reports an anomaly. In this way we can deal with the problem of having too many false positives and allow the user to set the sensitivity of the algorithm on his or her demand. The output from the whole module is: *(first anomaly window timestamp, last anomaly window timestamp)*.

In our setting, we used $window\ size = 32$, hamming smoothing window with $M = 12$, confidence interval under Gaussian ($error\ threshold = 0.99$) and queue size of $tolerance = 32 windows$.

D. Faulty pattern classification

Identifying the existence of an anomaly without providing any insight into its nature is of limited value. The user may be interested in detecting particular types of anomalies which reflect in the time series (e.g., incremental, mean shift, gradual increase, cylinder etc.). Here, we expect that an expert knows the types of patterns that commonly lead to service or component failure. Therefore, we provide a module based on one dimensional convolutional neural networks that, given as input a window of the event *response time* (e.g., 32 events), is able to classify into one of the user defined patterns described before. Convolutional Neural Networks (CNNs) [36] are common deep learning technique for image, signal processing, sequence and time series classification tasks. Typically, the architecture of the CNN consists combination of convolutional and max-pooling layers followed by softmax layer that distributes the probability for a given pattern. Recently, similar work for time series classification is found in [37].

Similar to the preprocessing module, the latest N points from time series are queried and utilized to represent the normal class. After that, we create a dataset which contains normal data preprocessed using the preprocessing module described and added to the augmented samples of predefined patterns (translation and adding small amount of noise) by the user.

The model architecture consists of three (convolutional, max-pooling) layers with dropout (0.5) regularization. The last layer, typical for multi-class classification, is fully connected with softmax function for computing the probability distribution over the classes. The convolutional networks are naturally invariant to translation, which makes them suitable for faulty pattern detection with sliding window over the time series. The network is trained using the data described and the model is saved and used for prediction. We used again Adam optimizer with optimal parameters obtained via cross validation ($learning\ rate = 0.001$, $number\ of\ epochs = 200$ and $batch\ size = 1000$). The classifier triggers when the *test-time prediction* detects an anomaly. The classifier module receives the output from *test-time prediction* and requests the particular time series within the provided anomalous time interval. Next, using the trained model, we map each sliding window to the predicted class and if the particular pattern is recognized the module will output the name of the class in which the pattern belongs and will flag the interval as anomalous.

V. Evaluation

The deep learning methods are implemented in Python using Keras [38]. The evaluation on the collected datasets were conducted on regular personal computer with the following specifications: GPU-NVIDIA GTX 1060 6GB, 1TB HDD, 256 SSD and Intel(R) Core(TM) i7-7700HQ CPU at 2.80GHz. In this section, we show evaluations of separate modules on two datasets. First, we show results on the experimental testbed system based on microservice architecture and later the evaluation on real large-scale production cloud data.

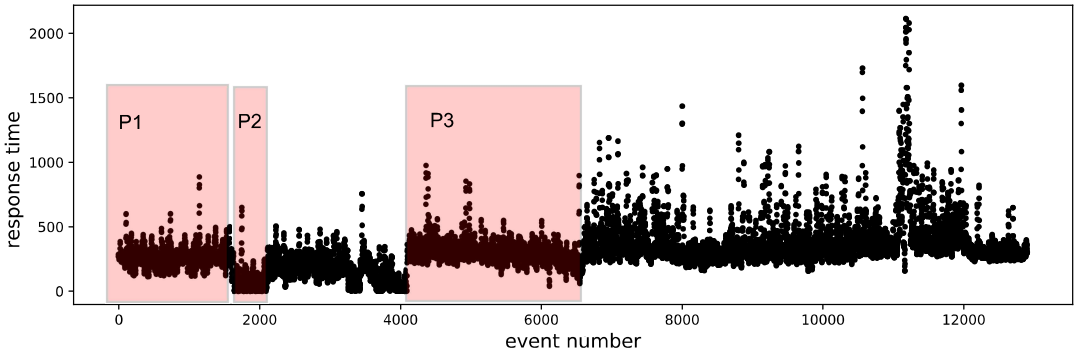


Fig. 4. Multiple distributions: We notice existence of multiple distributions that need to be learned as normal. The distribution differs from P1 compared to P2 and P3.

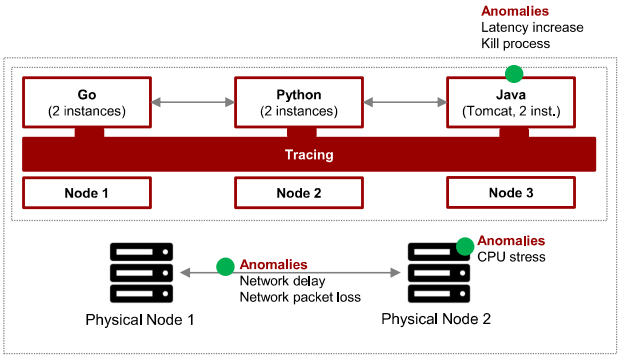


Fig. 5. Experimental microservice system architecture.

A. Experimental microservice system

We created an experimental microservice system to evaluate representative anomaly scenarios for microservice architectures. It allows fast preparation of experiments, known data format, and precise control on anomalies injected in the system. For the setup, we used 2 physical nodes and 3 virtual machines with tracing enabled, each of them running instances of Python, Go and Java applications respectively. The testbed architecture is shown in Figure 5. For the anomaly injections we used *stress-ng, traffic control, and simulator parameters*.

We injected timed anomalies in the physical network in form of delay and packet loss, physical node anomalies in form of CPU stress and event response time increase injected directly into the event. Thus, we created 6 different scenarios on different endpoints in the system described in following.

- Scenario 1: Baseline with no anomaly - represents the normal operation (no anomalies) of the system and is used to train the detection algorithms.
- Scenario 2: Increase service latency - profile 1 (injection of latency (1 second) for duration of 15 seconds).
- Scenario 3: Increase service latency - profile 2 (injection of latency (1, 5, 30 seconds) for duration of 30 seconds, 1 and 10 minutes on Nodes 1,2,3).
- Scenario 4: Network packet loss - Packet loss (10%, 20%, 30%) is injected on one of the network links for 1, 5, 10 minutes
- Scenario 5: Network delay - Network delay (1, 2, 3 sec) is injected on the network for 1, 5, 10 minutes.
- Scenario 6: Server process dies - One process is killed on node 1 and 3 for 1, 5, 10 minutes.

B. Dataset: production-cloud data

Even in small, controlled experimental setups, the amount of noise is high and the time series changes rapidly over time. This already opposes challenges for the anomaly detection algorithm. However, testing the approach on large-scale production cloud data is required to show the viability of the approach. The signal-to-noise ratio is even smaller since many components affect the *response time* of microservices, the time series evolves faster and changes its distribution over period of time while having also some stochastic behavior.

We collected the dataset in a period of four days from a real-world production cloud. We used only the attributes that are open-sourced by Zipkin [39], removing all the proprietary instrumentation. Therefore, from all of the attributes that one event has (around 80 in total, depending on the event), we extract only the *http URL* and the *response time*. The number of unique *http URLs* was 100168 and after clustering the total number of time series (*cluster IDs*/endpoints) was 143. Out of them, we selected three services of interest. The anonymized names along with the count of the samples is given in Table I

TABLE I
SELECTED CLUSTERS FOR ANALYSIS

ClusterID	Count
{host}/v1/{p_id}/cs/limits	12900
{host}/v1/{t_id}/cs/delete	2732
{host}/v2/{t_id}/servers/detail	6468

The time series, as shown in Figure 4, have high level of noise ([0ms, 2000ms] and [0ms, 4000ms]), several distributions changing over time, and no strong anomaly in both signals except in the neighbourhood of 11500^{th} data point in

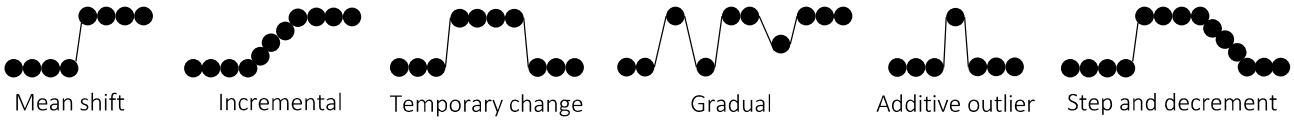
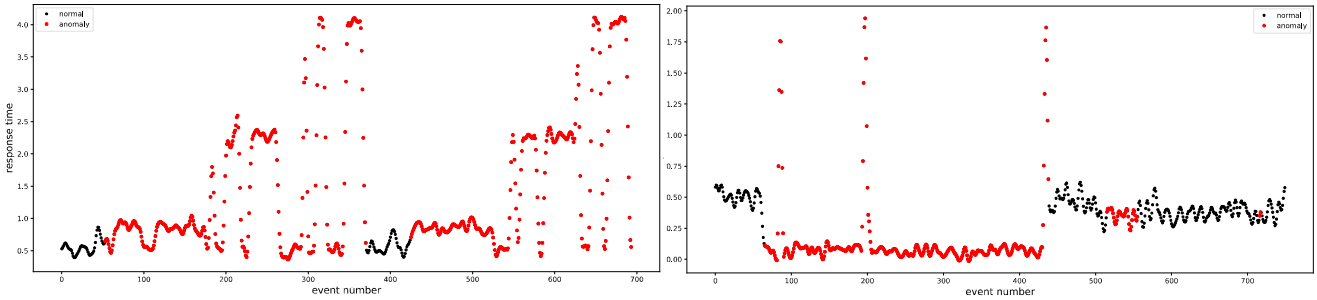


Fig. 6. Example of predefined patterns



(a) (b)

Fig. 7. Detected anomalies injected for different scenarios: (a) scenario 5, and (b) scenario 6

the figure. For the evaluation, we will use the part of the time series as normal scenario for training and then inject faults to check the model accuracy in the rest of the series. The presence of strong noise leads to low autocorrelation. That means, simple sequence learning without learning the distribution (with variational autoencoder) will result in learning only the running mean of the time series. Moreover, the distribution is skewed negatively. Therefore, the typical log transform of the data wont help for the model and it is omitted.

C. Results: variational recurrent model

In the following, we show the results obtained by the models on both datasets.

1) Results: microservice system: Accuracy of the unsupervised method for all 15 endpoints in our experimental testbed across 5 different scenarios are shown in Table II. We compute the accuracy in the following way. Let the number of injected anomaly events be denoted with T_i and the number of accurately detected anomalies be T_a, then the accuracy over the injected anomalies is computed as $accuracy = \frac{T_a}{T_i}$. The number of injected anomalies depends on each scenario and endpoint. The missing values in Table II mean that the anomaly did not affect those endpoints.

We note that the accuracy in normal system scenario is greater than 99% due to the tolerance module. We show scenario 5 and 6 graphically in Figure 7. We show that the method successfully flags almost all anomalous events. Overall, the results shown in the table and in the figure indicate that the combination of generative models like variational autoencoder with GRU units that extract temporal information achieves solid results.

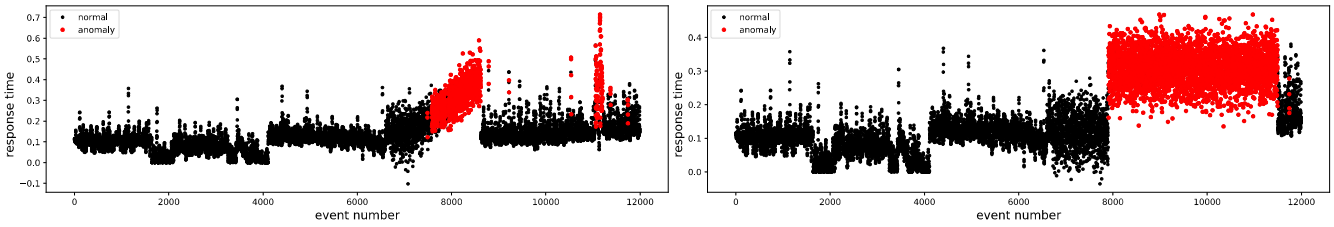
2) Results: production cloud data: Due to the low number of production-system errors, we injected several types of anomalies. We defined seven types of common anomalies:

TABLE II
ACCURACY FOR 15 ENDPOINTS IN 5 ANOMALY SCENARIOS

Clus. ID	S 2	S 3	S 4	S 5	S 6
1	-	85	95	98	99
2	-	-	99	98	-
3	-	-	96	99	96
4	-	99	-	-	-
5	-	-	100	98	97
6		98	-	-	86
7	-	95	98	97	-
8	-	98	-	-	-
9	-	92	91	99	100
10	90	95	95	99	98
11	-	96	-	-	-
12	85	-	83	99	97
13	95	98	-	94	99
14	99	96	-	-	98
15		97	95	98	100
average	85.5	95.3	94.6	97.9	97.0

samples from normal distribution with different mean, additive outlier, mean shift, step and decrement, incremental, temporary change and gradual. Some of the types of anomalies can be found in Figure 6.

The model is trained on each of the three endpoints and for each of the endpoints we compute the accuracy of detection for each of the patterns injected in the time series. To test the robustness of the algorithm, we evaluated it for several different augmentations of the original patterns, including translation, increasing the response time (e.g., $RTi = 0.2$ means setting the response time of event to 0.2 of the maximum value) and change of the size of anomaly (e.g., in gradual increase, $size = 10$ means that the increase from amplitude A to amplitude B is gradual over 10 events/data points). In Table III, we show the aggregated results. The table summarizes the different anomaly patterns and the needed minimum values



(a) (b)

Fig. 8. Example of successfully detected anomalies injected in {host}/v1/{p_id}/cs/limits. Gradual and mean shift anomalies are injected in (a) and (b) respectively.

for the parameters $size$ and RTi which lead to a detection of the corresponding anomaly type. Further, in Figure 8a and 8b we visually illustrate some of the patterns detected in the series. We notice that the algorithm successfully detects the three types of anomalies injected.

TABLE III
RESULTS FOR VARIATIONAL RECURRENT MODEL

Pattern name	Parameters
additive outlier	$RTi > 0.25$
normal_Mean	$RTi > 0.2$
temporary change	$RTi > 0.25$
gradual	$RTi > 0.3,\ size > 10$
mean shift	$RTi > 0.2$
step and decrement	$RTi > 0.3,\ size > 10$
incremental	$RTi > 0.3,\ size > 10$

3) Results: faulty pattern classification: The module can be evaluated separately from the rest of the solution, since data and predefined patterns as described. The dataset consists of 15 different types of patterns similar to the ones shown in Figure 6. In practice, the user has the option to define own patterns.

Furthermore, besides the patterns shown in Figure 6, we produce augmentations to enrich the dataset. The augmentations are produced by using: horizontal shifts, adding small amount of noise and amplitude shifts.

We evaluated the algorithm to see the performance and its limits when it comes to the level of noise in the signal and the accuracy of classification. We achieved 100% accuracy in data with no additional noise added, 80% and 48% accuracy when Gaussian noise was added with $\sigma = [0.05 \text{ and } 0.1]$ respectively. The convolutional neural network model accurately classifies the tested anomaly patterns, with expected lower accuracy obtained in noisy patterns.

TABLE IV
PERFORMANCE EVALUATION IN TRAINING

#windows	ms/window
60000	0.29
27000	0.28
9000	0.53
1000	1.25

4) Performance evaluation: In real-world production systems, the performance of the model in training and prediction time is very important. Having large amounts of traces and events generated in short period of time, requires fast prediction time and timely detection of anomalies. For that reason, we evaluate the performance of the approach. We show the results in Tables IV and V. Lastly, imposing industrial requirements ($\text{prediction time} \leq 10\text{ms}$) and meeting the criteria for performance proves that the approach is fast enough to be used in production setting. In streaming test-time prediction we achieve performance of 6.64ms per predicted window of points. Of course, the prediction times can differ with reducing or expanding the *window size*, but it is still within the limits of the necessary requirements.

TABLE V
PERFORMANCE EVALUATION IN TEST-TIME PREDICTION

#windows	ms/window
1500	0.22
1000	0.28
500	0.53
100	1.25

VI. CONCLUSION AND FUTURE WORK

This paper deals with an important and growing challenge: the automation of operation and maintenance tasks of planet-scale IT infrastructures. We experimentally demonstrate the advantages of combining GRUs (simplified LSTMs) with variational autoencoders (AEVB) – two deep learning models – for learning multiple, complex data distributions underlying time series data generated by distributed tracing systems. Our investigation on experimental and real-world production data with artificially injected anomalies showed that our approach reaches accuracy greater than 90%, prediction time lower than 10ms, and robust classification of detected anomalies. The tracing data was generated by an experimental microservice application and by a planet-scale cloud infrastructure. These high levels of accuracy open a new door in the field of AIOps. Namely, the approach can be extended to also consider the structure of distributed traces, use knowledge from cross-event and cross-trace relations, and analyze structural trace anomalies using complete trace information. These achievements can ultimately support the development of zero-touch AIOps solutions for the automated detection, root-cause analysis and remediation of IT infrastructures.

REFERENCES

[1] F. Schmidt, A. Gulenko, M. Wallschlger, A. Acker, V. Hennig, F. Liu, and O. Kao, "Iftm - unsupervised anomaly detection for virtualized network function services," in *2018 IEEE International Conference on Web Services (ICWS)*, July 2018, pp. 187–194.

[2] A. Gulenko, F. Schmidt, A. Acker, M. Wallschlager, O. Kao, and F. Liu, "Detecting anomalous behavior of black-box services modeled with distance-based online clustering," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 912–915. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00134

[3] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[4] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi *et al.*, "Canopy: an end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 34–50.

[5] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1973430.1973450

[6] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "Wap5: black-box performance debugging for wide-area systems," in *Proceedings of the 15th international conference on World Wide Web*. ACM, 2006, pp. 347–356.

[7] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 13–24.

[8] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 74–89, 2003.

[9] T. Gschwind, K. Eshghi, P. K. Garg, and K. Wurster, "Webmon: A performance profiler for web transactions," in *Advanced Issues of E-Commerce and Web-Based Information Systems, 2002.(WECWIS 2002). Proceedings. Fourth IEEE International Workshop on*. IEEE, 2002, pp. 171–176.

[10] P. Barham, R. Isaacs, and D. Narayanan, "Magpie: online modelling and performance-aware systems," in *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*. USENIX, May 2003, pp. 85–90. [Online]. Available: https://www.microsoft.com/en-us/research/publication/magpie-online-modelling-and-performance-aware-systems/

[11] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.

[12] F. Bezerra and J. Wainer, "Algorithms for anomaly detection of traces in logs of process aware information systems," *Information Systems*, vol. 38, no. 1, pp. 33–44, 2013.

[13] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, "Recurrent neural network attention mechanisms for interpretable system log anomaly detection," *arXiv preprint arXiv:1803.04967*, 2018.

[14] *Mining Invariants from Console Logs for System Problem Detection*. USENIX, June 2010. [Online]. Available: https://www.microsoft.com/en-us/research/publication/mining-invariants-from-console-logs-for-system-problem-detection/

[15] D. Battre, O. Kao, and D. Warneke, "Evaluation of network topology inference in opaque compute clouds through end-to-end measurements," in *2011 IEEE 4th International Conference on Cloud Computing*, July 2011, pp. 17–24.

[16] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[17] M. Joshi, R. Agarwal, and V. Kumar, "Mining needle in a haystack: classifying rare classes via two-phase rule induction," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 91–102, 2001.

[18] M. V. Joshi, R. C. Agarwal, and V. Kumar, "Predicting rare classes: Can boosting make any weak learner strong?" in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 297–306.

[19] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Special issue on learning from imbalanced data sets," *ACM Sigkdd Explorations Newsletter*, vol. 6, no. 1, pp. 1–6, 2004.

[20] H. Fichtenberger, M. Gillé, M. Schmidt, C. Schwiegelshohn, and C. Sohler, "Bico: Birch meets coresets for k-means clustering," in *European Symposium on Algorithms*. Springer, 2013, pp. 481–492.

[21] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 7, pp. 881–892, 2002.

[22] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 2008, pp. 413–422.

[23] L. M. Manevitz and M. Yousef, "One-class svms for document classification," *Journal of machine Learning research*, vol. 2, no. Dec, pp. 139–154, 2001.

[24] O. Vallis, J. Hochenbaum, and A. Kejariwal, "A novel technique for long-term anomaly detection in the cloud," in *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=2696535.2696550

[25] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long short term memory networks for anomaly detection in time series," in *Proceedings*. Presses universitaires de Louvain, 2015, p. 89.

[26] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2018, pp. 187–196.

[27] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom, "Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining*, ser. KDD '18. New York, NY, USA: ACM, 2018, pp. 387–395. [Online]. Available: http://doi.acm.org/10.1145/3219819.3219845

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[29] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *International Conference on Learning Representations (ICLR)*, 2014.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, p. 533, 1986.

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[32] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8)*, 2014.

[33] J. Chung, K. Kastner, L. Dinh, K. Goel, A. C. Courville, and Y. Bengio, "A recurrent latent variable model for sequential data," in *Advances in neural information processing systems*, 2015, pp. 2980–2988.

[34] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 448–456.

[35] A. Nuttall, "Some windows with very good sidelobe behavior," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 1, pp. 84–91, February 1981.

[36] Y. LeCun, Y. Bengio *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[37] B. Zhao, H. Lu, S. Chen, J. Liu, and D. Wu, "Convolutional neural networks for time series classification," *Journal of Systems Engineering and Electronics*, vol. 28, no. 1, pp. 162–169, 2017.

[38] F. Chollet *et al.*, "Keras," https://keras.io, 2018.

[39] OpenZipkin, "openzipkin/zipkin," 2018. [Online]. Available: https://github.com/openzipkin/zipkin