Tampere University

Pino' Surace

ANOMALY DETECTION IN CLOUD-NATIVE SYSTEMS

Faculty of Information Technology and Communications Sciences
Master of Science Thesis
November 2019

ABSTRACT

Pino' Surace: Anomaly Detection in Cloud-Native systems
Master of Science Thesis
Tampere University
Major: Software Engineering - Web & Cloud
November 2019

In recent years, microservices have gained popularity due to their benefits such as increased maintainability and scalability of the system. The microservice architectural pattern was adopted for the development of a large scale system which is commonly deployed on public and private clouds, and therefore the aim is to ensure that it always maintains an optimal level of performance. Consequently, the system is monitored by collecting different metrics including performance-related metrics.

The first part of this thesis focuses on the creation of a dataset of realistic time series with anomalies at deterministic locations. This dataset addresses the lack of labeled data for training of supervised models and the absence of publicly available data, in fact the data are not usually shared due to privacy concerns.

The second part consists of an empirical study on the detection of anomalies occurring in the different services that compose the system. Specifically, the aim is to understand if it is possible to predict the anomalies in order to perform actions before system failures or performance degradation. Consequently, eight different classification-based Machine Learning algorithms were compared by collecting accuracy, training time and testing time, to figure out which technique might be most suitable for reducing system overload.

The results showed that there are strong correlations between metrics and that it is possible to predict the anomalies in the system with approximately 90% of accuracy. The most important outcome is that performance-related anomalies can be detected by monitoring a limited number of metrics collected at runtime with a short training time. Future work includes the adoption of prediction-based approaches and the development of some tools for the prediction of anomalies in cloud native environments.

Keywords: Anomaly Detection, Machine Learning, Empirical Study, Time series, Data Generation

PREFACE

This thesis is the final outcome of a work which lasted almost ten months. I would like to thank all the people who supported me during the thesis project and my studies at the Tampere University.
In would like to thank the examiners of this thesis, Assistant Prof. Davide Taibi and Post-doctoral Researcher Valentina Lenarduzzi for providing valuable feedback and guidance during this work.
Finally, I would like to thank those who are close to me, my family and my girlfriend, who gave me a great deal of support.

Special thanks to my father, who made this all possible.

Tampere, 10th November 2019

Pino' Surace

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Program Interface
AUC	Area Under the Receiver Operating Characteristic Curve
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MLP	Multi-Layer Perceptron
PCA	Principal component analysis
REST	Representational State Transfer
ROC	Receiver Operating Characteristics

1 INTRODUCTION

In recent years, microservices have gained popularity due to their benefits which include increased maintainability and higher scalability of the system. Moreover, microservices help to decrease technical debt [60] and allow teams to develop and deploy their respective services independently. The motivations why companies are migrating to microservices and the processes adopted for the migration are reported in [67]. Architectural patterns that should be adopted in microservices-based systems are reported in [68] and [69] while anti-patterns are reported in [66] and [70]. An “interesting” process to identify the possible “cuts” for microservices from monolithic systems is proposed in [72] and then extended with a measurement framework to evaluate the quality of the decomposition in [71]. The microservice architectural pattern was adopted for the development of a large scale system which is composed of several microservices running on top of Kubernetes and communicating using a lightweight message bus (Apache Kafka [39]), as shown in Figure 1.1.

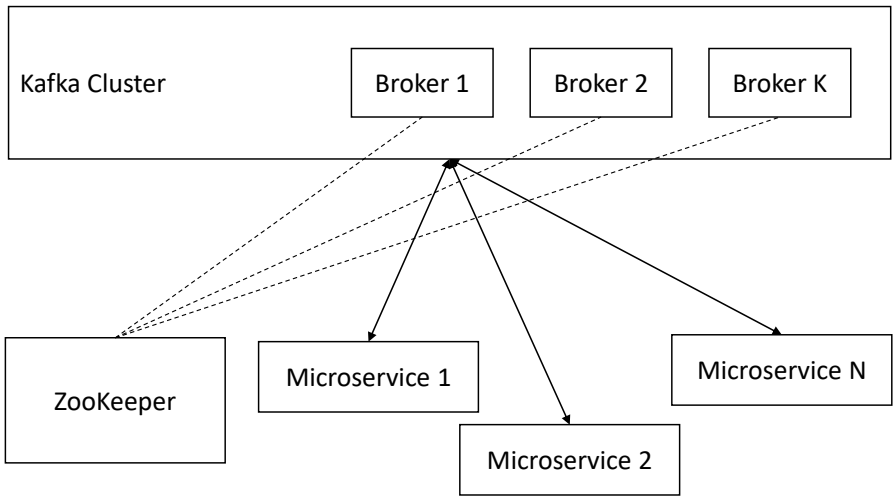


Figure 1.1. *System architecture*

The size of the system requires to have multiple Kafka brokers and therefore Zookeeper is needed to coordinate the different Kafka instances. Unlike monolithic architectures, this system consists of multiple components (orchestrators, load balancers, message buses, etc.) that could fail and the services are deployed on different machines. In such a complex system, runtime failures are unavoidable [14] and must be kept under control.The

system is commonly deployed on public and private clouds. Since private clouds often have limited resources, the aim is to ensure that the system always maintains an optimal level of performance. All the services composing the system are actively monitored using Prometheus [7], which collects 168 different metrics including performance-related metrics, hardware failures, and metrics related to the communication between services, such as throughput and message lags. The purpose of monitoring is not only to check that all the cloud-native services are up and running, but also to ensure that the customers' private clouds are not overloaded and to avoid degradation of the overall performance due to anomalies. This could be achieved by attempting to detect whether resources are used completely, so that services can respond on time to all the requests, while ensuring that the communication between services occurs with a very small lag.

The fist part of this study focuses on the creation of a dataset of realistic time series with anomalies at deterministic locations. This dataset addresses the lack of labeled data for training of supervised models and at the same time the absence of publicly available data, in fact the data are not usually shared due to privacy concerns.

The second part consists of an empirical study on the detection of anomalies occurring in the different services that compose the system. Specifically, the aim is to understand if it is possible to predict the anomalies in order to perform actions before system failures or performance degradation.

For example, a service might use an abnormal amount of memory or processors which will not enable other services to run properly. As another example, a consumer service might stop sending fetch requests to the broker, which means the service could be stalled or dead.

The most challenging issue related to anomaly detection is to understand which metrics can be considered *benign anomalies*, i.e., anomalies that occur in the case of unusual but correct execution of the system and do not lead to any failure or performance issue [37].

Different anomaly detection techniques have been proposed in literature. The data-driven techniques are based on the analysis of data collected at runtime and are designed to predict anomalies in complex systems based on abnormal system behaviour [31]. In addition, researchers have proposed both supervised and unsupervised Machine Learning techniques for anomaly detection. The unsupervised models are trained based on data from the correct execution of a system, however, they are less accurate than supervised ones. Instead the supervised models train the model under consideration of both normal and failing execution data [37][61].

Besides accuracy, another issue that needs to be considered by an anomaly detection mechanism is the training time required by Machine Learning algorithms [50]. The system produces a huge amount of data every day, and training the system on this data could become very expensive in terms of time and resources required. Since the aim is to reduce the system overload in the customers' clouds, it is unsuitable to run an anomaly detection system that will consume more resources than the monitored system itself.

Therefore, in order to understand which Machine Learning technique might be most suitable for reducing system overload, eight different Machine Learning algorithms were compared.

After that, statistical and machine learning approaches have been applied for the identification of the most important metrics and for the identification of redundant information in the metrics.

This work will contribute to the body of knowledge of industrial experience on anomaly detection. It will help companies that are working with cloud-native systems based on similar technologies as well as researchers to understand how the different techniques perform and to conduct empirical studies in the industry. Moreover, the dataset created provides labeled metrics that could be used to train machine learning models, to perform studies and to compare results on the same publicly available dataset.

The remainder of the thesis is divided into seven chapters. In Chapter 2, the basic concepts underlying this work and the main technologies are described. In Chapter 3 related research on anomaly detection and time series data generation are reported. In Chapter 4 the data collection process and the creation of the dataset are presented. Chapter 5 describes the case study design, research questions, metrics, hypotheses, and the study context. In Chapter 6, the achieved results are presented. Finally, in Chapter 7 results and their limitations are discussed and in Chapter 8 conclusions are drawn.

2 BACKGROUND

2.1 Cloud Native Systems

Cloud-Native systems [22] are applications built on private or public cloud and they are characterized by multiple features which include horizontal scaling, vertical scaling and flowing fault prone-infrastructure. Horizontal scaling means that data are accessed globally from the internet and replicated so that the latency of services is reduced. Instead, vertical scaling signify that data are accessed simultaneously by many clients. Moreover Cloud-Native systems are characterized by a flowing fault-prone infrastructure which means that things break often because of the large horizontal scale. For this reason, security is part of the architecture design . In addition, upgrade and test occur without interrupting normal operations.

The first type of Cloud-Native systems developed was Infrastructure as a Service (IaaS). It replaced infrastructures hosted on-premise by instances running on the cloud. This solution was not advantageous for massive applications due to security and scalability issues.

In 2010, Platform as a Service (PaaS) was developed. It allowed the abstraction of data management and event handling.

In 2013, the microservice pattern was developed. In this pattern, the scalability and reliability goals were achieved by dividing the applications into small units that are managed, replicated, scaled, upgraded, and deployed independently from each other. Each microservice has a single function and a limited context with limited responsabilities and dependencies. Microservices are designed to be fluid and restart after each failure. For this reason, they must be stateless, and the state of the application is stored by using some persistence layer. Microservices communicate using REST APIs and RPC mechanisms and they are packaged into containers so that they can be easily started, stopped and migrated. One of the most used standards for containerizing applications is Docker. To take care of scaling, replicating and restarting failed containers automatically, Cloud-Native application deployment services have been developed, such as Kubernetes, Docker Swarm and OpenStack.

2.1.1 Apache Kafka

Apache Kafka [51] [74] [45] is an open-source publish/subscribe messaging system where data are stored durably and distributed to assure scalability and reliability. The unit of data used by Kafka is a message, i.e., an array of bytes. Messages can have different formats such as Javascript Object Notation (JSON) and Extensible Markup Language (XML) that are written into Kafka in collections of messages called batches. The messages are organized into topics which are equivalent to folders in a filesystem. Topics are divided into different partitions which can be hosted in different machines to provide redundancy and scalability.

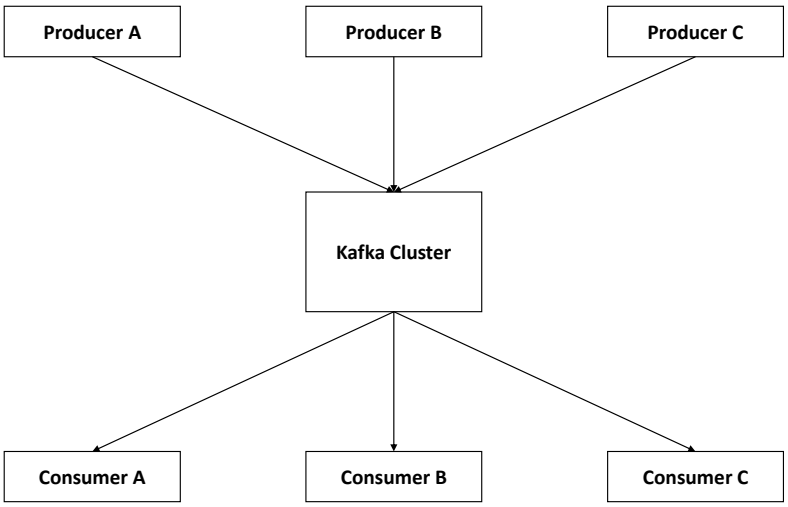


Figure 2.1. *Apache Kafka*

As shown in Figure 2.1, Kafka interacts with two kinds of clients: producers and consumers. Producers write new messages to a specific topic, while consumers subscribe to some topics and read messages in the order they have been produced. Kafka is composed of multiple servers called brokers. Each broker receives the messages from the producers and stores them on a disk. Usually, brokers are organized in clusters, where one of them is the controller. The controller takes care of the administration of the cluster by assigning partitions to broker and monitoring for broker failures. Kafka has many features that make it superior to many other producer/consumer messaging systems such as multiple consumers, multiple producers, strong retention, scalability, and high performance under high load.

2.1.2 Zookeeper

Apache ZooKeeper [38] is an open source project and it was developed to handle the coordination tasks for distributed systems such as master server election, group membership management and metadata management. These tasks can be of two types: cooperation tasks, when processes need to do something together, and contention when two processes cannot work in parallel, so one must wait for the other. ZooKeeper exposes a simple API that provides numerous benefits such as high consistency, ordering and durability, simpler implementation synchronization primitives and simpler handling of concurrency tasks.

2.1.3 Kubernetes

Kubernetes [29] is an orchestrator for the deployment of containers and realiable and scalable cloud systems. It was developed at Google and nowadays it is developed and maintained by a large open source community. Most of the cloud distributed systems, because of their nature should have high availability and scalability, in fact the system should not crash even if a part of it would fail and it should increase its capacity when the resources are not enough. Containers orchestrated by Kubernetes allow to build realiable and scalable systems in addition to several other benefits.

One of those benefits is velocity, measured in terms of features shipped while maintaining high available service. In fact containers and Kubernetes can provide tools to move quickly while staying available. This is possible due to the three fundamental concepts: immutability, declarative configuration, and self-healing. Immutability refers to the idea of having immutable containers. This means that instead of having incremental changes of the image, a new image is built. In this way a new image will always replace the former one with a single operation and in case of a problem it is possible to rollback to the previous image easily unlike with incremental updates. Declarative configuration refers to the declaration of the state of the system in contrast to imperative configuration where the desired state of the system is achieved by executing a set of instructions. In Kubernetes everything is a declarative configuration object, therefore it is very easy to declare the desired state of the system and the rollback is easy. Moreover, it helps with the source control, code reviews and unit tests. Self healing refers to the property of Kubernetes which constantly ensures that the current state is exactly as the desired state by taking actions such as killing or restarting containers. In this way the developers should not waste time by taking care and monitoring that everything is working, but instead they can focus on creating value.

In addition, Kubernetes accomodates scaling software and teams by encouraging a decoupled architecture, where each service is independent from the others and communicate using APIs and load balancers. Kubernetes provides multiple abstractions and APIS

for the developement of decoupled microservice architectures, such as Pods, load balancing, naming, service discovery and namespaces. Decoupling via APIs allows to scale the teams because each team can focus on a microservice without need of cross-team communication. Decoupling via load balancers allows to scale the capacity without touching other layers of the service. Furthermore, by abstracting the infrastructure, Kubernetes separates developers from specific machines and allows high portability.

2.1.4 Prometheus

Prometheus [7] is an open-source alerting and monitoring system developed by SoundCloud. It is widely used and it has many features such as a multidimensional data model, multiple dashboards modes support, HTTP based time series collection and gateway supported pushing of time series. One of the most important benefits Prometheus provides is the high reliability on microservices monitoring.

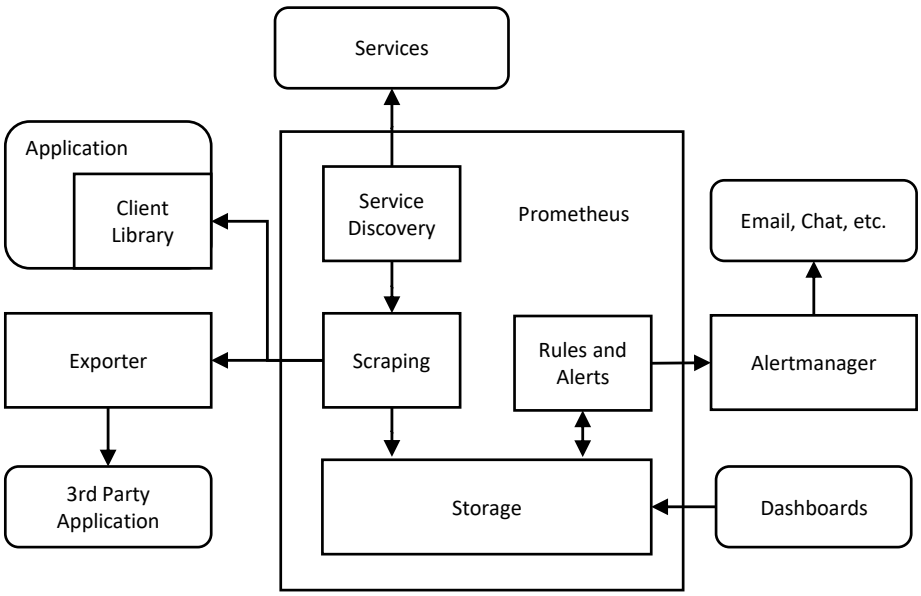


Figure 2.2. *Prometheus Architecture*

Figure 2.2 shows the architecture of Prometheus and its components. The Client Library takes care of instrumenting and producing metrics in the Prometheus text format in response to HTTP requests. The Exporter is a software that runs close to another software and gets the requests from Prometheus, gathers the metrics from the other software and returns them to Prometheus in the correct format. It is used where it is not possible to use directly the client library in the software. The Service Discovery is used to discover applications in dynamic environments such as cloud systems. The Scraping sends and HTTP request called a scrape to fetch the metrics. The response is parsed and stored

into Storage. Scrapes happen regularly, usually every 10 or 60 seconds. The Storage is a customized database that provides high reliability. Dashboards can be produced by leveraging the HTTP APIs provided by Prometheus. Rules and Alerts are PromQL expressions that are evaluated regularly and in case of alerts they will be raised and sent to the Alertmanager which receives alerts from the Prometheus server and turns them as notifications such as email or chat applications like Slack.

2.2 Anomaly detection

Anomaly detection [11] addresses the problem of finding patterns in data with unexpected behavior, called anomalies or outliers (an example can be seen in Figure 2.3). Anomaly detection is applied to multiple domains such as fraud detection, intrusion detection and health care. Because each domain has different data and different approaches, during the years many different techniques have been developed and they can be summarized into six categories: classification-based, clustering-based, nearest neighbor-based, statistical techniques, information theoretic techniques and spectral techniques. The choice of the techniques is affected by multiple factors which include nature of input data, type of anomalies, data labels and output. Data types can be divided into three categories which include binary data, categorical data and continuous data. There are multiple types of anomalies which include point anomalies, collective anomalies and contextual anomalies. Point anomalies occur when instances of data can be considered abnormal respect to the others. Collective anomalies occur when a subset of related data instances is abnormal respect to the entire data set. Contextual anomalies take place when the data is anomalous into a specific context.

Based on data labels there are three categories of anomaly detection techniques. Supervised, when labels are available for both normal and anomalous data. Semisupervised, when labels are available only for the normal class. Unsupervised, when there are not labels, but it is assumed that normal data are more frequent in the dataset. There are two different types of outputs: scores and labels. Scores will give a percentage to which the data instance is considered anomalous. Instead, labels will identify data instances as anomalies or normal without any accuracy degree.

Because of the nature of data, one-class classification-based techniques have been used in this study. These approaches are the most used ones in anomaly detection, consist of algorithms that learn a model from a set of labeled data instances (training) and classify test data using the learned model. The main advantages of the Classification-based techniques are a fast testing phase and the availability of powerful algorithms for classification. The main disadvantages are the need for accurate labels and that the output is only a label and therefore it is not possible to have a score.

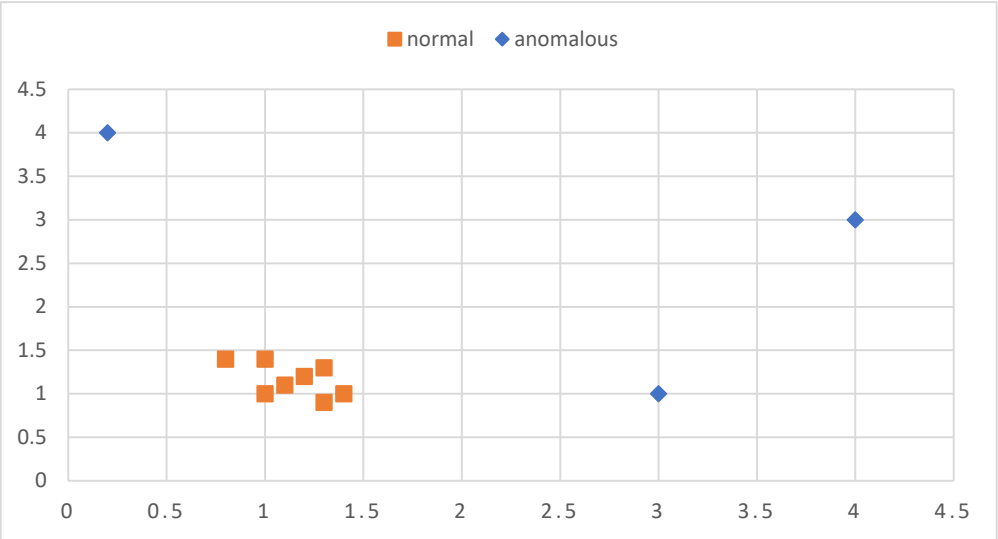


Figure 2.3. *Graphic representation of anomalies*

2.3 Machine Learning Techniques

In this study eight Machine Learning models were used for classification and were compared.

2.3.1 Logistic Regression

One of the simplest algorithms used in Machine Learning is *Logistic Regression* [16]. In contrast to linear regression, which is used to predict a numerical value, Logistic Regression is used for predicting the category of a sample. In particular, a binary Logistic Regression model is used to estimate the probability of a binary result (0 or 1) given a set of independent variables. Once the probabilities are known, they can be used to classify the inputs into one of the two classes based on their probability of belonging to either of the two. Like all linear classifiers, Logistic Regression projects the P-dimensional input $\mathbf{x}$ into a scalar by a dot product of the learned weight vector $\mathbf{w}$ and the input sample: $\mathbf{w} \cdot \mathbf{x} + w_0$, where $w_0 \in \mathbb{R}$ the constant intercept. To have a result which can be interpreted as a class membership probability—a number between 0 and 1—Logistic Regression passes the projected scalar through the logistic function (sigmoid). This function, for any given input x, returns an output value between 0 and 1. The logistic function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Finally, the class probability of a sample $\mathbf{x} \in \mathbb{R}^P$ is modeled as

$$Pr(c = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + w_0)}}.$$

Logistic Regression is trained through maximum likelihood: the model's parameters are estimated in a way to maximize the likelihood of observing the inputs with respect to the parameters $\mathbf{w}$ and w_0. This model was chosen to be used as baseline due to its simplicity and its easy implementation: by requiring few computational resources, it is easy to implement and fast to train. Moreover, it doesn't need the inputs to be scaled nor it needs to be tuned.

2.3.2 Decision Tree

One of the most frequently used models in Machine Learning is a *Decision Tree* classifier [9]. The tree structure is characterized by multiple nodes: the *root node* and the *internal nodes*, which represent the inputs, and a series of *leaves*, which correspond to the outputs. All these nodes are connected via branches. A specific path through the branches represents an output.

2.3.3 Random Forest

Decision trees tend to have overfitting problems, as they cannot learn to generalize the data properly. For this reason, also a *Random Forest* model [8] was tested. This is an ensemble model, as it uses a set of simpler models to solve the assigned task. In this specific situation, it uses multiple decision trees. Each decision tree is trained on a different subset of the data. The results of all the decision trees in the Random Forest are averaged to obtain a single output.

2.3.4 Extremely Randomized Trees

To increase the randomization degree of the Random Forest algorithm, the *Extremely Randomized Trees* (ExtraTrees) model [23] was used: Besides randomly splitting the data for each of the individual trees, the optimal split for each node is also randomized in the ExtraTrees model. This model is less computationally expensive while its generalization capabilities are increased.

2.3.5 AdaBoost

Another class of ensemble algorithms used in this study is based on *boosting* [63]. One of these models is AdaBoost [20]. AdaBoost creates individual decision trees sequentially and assigns a weight to each training set sample, which is modified during the training. It keeps on creating decision trees and adjusting the weights until the model can no longer be improved in terms of accuracy.

2.3.6 Gradient Boosting

Besides AdaBoost, the *Gradient Boosting* algorithm [21] was included in the analysis. Unlike AdaBoost, it grows one tree at a time in order to minimize loss. This process continues until the loss function can no longer be improved.

2.3.7 XGBoost

Due to the heavy computational expense of training the Gradient Boost model, we also considered XGBoost [13]. This model is merely a better performing implementation of the Gradient Boosting algorithm, allowing faster computation and easier parallelization. This allows it to perform better and to be more easily scaled to bigger datasets.

2.3.8 Multi-Layer Perceptron

The last classifier used is based on a *Multi-Layer Perceptron* (MLP) [58]. A classifier based on MLP is a supervised learning model that, through training, learns a non-linear function in order to classify a set of inputs. To do so, it uses *backpropagation*: During each training cycle, the error of the output is propagated backwards to update the weight of the nodes of the MLP. This is done until the error in the output is minimized.

3 RELATED WORK

This chapter describes extensively the related research conducted respectively on anomaly detection in the cloud and on time series generation.

3.1 Anomaly detection

Anomaly detection has been investigated in several domains in recent years by applying probabilistic [26] and statistical [35] approaches.

Hochenbaum et al. [30] proposed two statistical approaches for automatic anomaly detection in cloud infrastructure data. Indeed, the proposed methods apply statistical learning for anomaly detection in system and application metrics. Seasonal decomposition is used to filter trends and seasonal components of the time series. Moreover statistical metrics such as median and median absolute deviation (MAD) are employed to accurately detect anomalies, despite of seasonal spikes.

Solaimani et al. [65] proposed a Chi-square based anomaly detection approach on heterogeneous data by leveraging the high processing power of Apache Spark.

Smrithy et al. [64] developed an algorithm based on Kolmogorov-Smirnov goodness of fit test for anomaly detection of access requests at runtime in cloud enviroments.

Wang et al. [77] proposed statistical techniques for online anomaly detection. The proposed approaches are lightweight and based on Tukey and Relative Entropy statistics.

Roy et al. [57] developed PerfAugur, a system for the detection of anomaly behaviors using data mining algorithms in service logs.

Statistical models perform well in identification of anomalies and they do not require a big amount of data for training models. Despite this, the main obstacle of these techniques is the production of biased results in case of inaccurate hypothesis on the data. This leads to many false positives and makes statistical approaches not suitable for real applications.

On the other hand, machine learning approaches are capable of inferring distribution of normal and anomalous behaviors, and determine anomalies by using supervised, semi-supervised, unsupervised, or deep learning techniques [32]. Supervised techniques need labeled data for normal and anomalous behavior and can be extremely precise, but perform poorly in detecting anomalous behaviors not previously encoded in the train-

ing set. Unsupervised techniques, instead, can infer patterns encoded in the unlabeled data, but they often detect anomalies not related to failures. For this reason, they need a big amount of data and long training process to increase the precision.

Ahmed et al. [2] proposed a sequential anomaly detection technique based on the kernel version of the recursive least squares algorithm. This approach can be used effectively also for multivariate data.

Lakhina et al. [46] presented an anomaly detection approach based on the division of the high-dimensional space represented by a set of metrics into disjoint subspaces corresponding to normal and anomalous behaviors. To perform the separation, Principal Component Analysis has been employed successfully.

Ibidunmoye et al. proposed two methods, PAD [33] and BAD [34], based on statistical analysis and kernel density estimation (KDE) applied to unbalanced data. The performances of these methods are affected by the window size used for the estimation.

Thill et al. [75] proposed SORAD, an anomaly detection approach based on regression techniques.

Ahmad et al. [1] presented a real-time anomaly detection algorithm based on Hierarchical Temporal Memory (HTM) and suitable for spatial and temporal anomaly detection in predictable and noisy environments.

Hochenbaum et al. [30] developed two statistical approaches for anomaly detection in cloud infrastructure data. Their first method called Seasonal-ESD combines seasonal decomposition and the Generalized ESD test, for anomaly detection.The second approach called Seasonal-Hybrid-ESD (S-H-ESD) adds statistical measures such as median and median absolute deviation (MAD) to the previous algorithm.

Mi et al. [49] developed CloudDiag, a tool for performance anomaly detection based on unsupervised learning.

Dean et al. [17] developed UBL, a distributed and scalable anomaly detection system for Infrastructure as a Service (Iaas) cloud environments based on unsupervised learning. It leverages the power of Self-Organizing Map (SOM) to detect performance-related anomalies to provide suggestions on possible issues.

Tan et al. [73] developed PREPARE, a performance-related anomaly prevention system for virtualized cloud computing infrastructure. It combines attribute value prediction with supervised anomaly classification methods to perform resource scaling for performance anomalies prevention.

Guan et al. [24] implemented an unsupervised proactive failure management framework for cloud infrastructures based on a combination of Bayesian models to perform anomaly detection with high true positive rate and low false positive rate.

Gulenko et al. [25] proposed an event-based approach to real-time anomaly detection in cloud-based systems with a specific focus on the deployment of virtualized network

functions. They applied both supervised and non-supervised classification algorithms, obtaining good results in the idenfication of anomalies.

Monni et al. [50] proposed an energy-based anomaly detection tool (EmBeD) for the cloud domain. The tool is based on a Machine Learning approach and is able to reveal failure-prone anomalies at runtime. EmBeD exploits the system behavior using the raw metric data, classifying the relationship between anomalous behavior and future failures with a good level of accuracy (in terms of very few false positives). Moreover, Monni et al. [50] also defined an energy-based model to capture failure-prone behavior without training with seeded errors. They identified important analogies regarding the nature of complex software systems, complex physical systems, and complex networks.

Sauvanaud et al. [61] applied machine learning approaches such as Neural Networks, Naive Bayes, Nearest Neighbors, and Random Forest for anomaly detection at metric level.

3.2 Time series generation

Data generation has been applied to different domains and multiple techniques have been adopted to achieve good results. For example, Alzantot et al. [3] proposed a deep learning based architecture for sensory data generation. Ledig et al. [48] presented SRGAN, a generative adversarial network (GAN) for photo-realistic high resolution images.

Reed et al. [56] used a GAN model for the generation of images of birds and owers from detailed text descriptions.

Bowman et al. [6] introduced an RNN-based variational autoencoder for text generation.

The first studies have been applied mostly to images, but recently promising results have been presented in studies that apply similar techniques to the time series in different domains.

Hartmann et al. [28] proposed an approach based on GAN trained on a 128-electrode electroencephalograph (EEG) data set for the generation of time series EEG data.

Esteban et al. [18] proposed a technique for time series generation combining time series sinusoidal data and physiological metrics such as oxygen saturation, respiratory rate, heart rate, and mean arterial pressure. This method can generate sequences of 30 data points by adopting recurrent conditional generative adversarial networks (RCGAN).

Brophy et al. [10] proposed a simplified approach for time series data generation by leveraging image-based GAN techniques.

Hahmann et al. [27] proposed a feature-based generation method for large-scale time series.

Forestier et al. [19] introduced a framework for generating synthetic time series under

Dynamic Time Warping.

Iftikhar et al. [36] proposed a supervised machine learning approach for meter data generation. It has been developed using Apache Spark it allows generation of scalable data sets on a cloud infrastructure.

Kang et al. [40] developed a method for time series data generation with controllable characteristics. This technique allows to explore all the feature space so that it is possible to generate time series similar to the original or generate time series with particular features. This approach is very useful for generating data for training models so that they do not over-fit to the original data set.

Kegel et al. [42] [43] presented a general and simple technique for the generation of what-if scenarios on time series data. This method gathers descriptive features from data and allows the user filtering and modification operations.

Kegel et al. [44] implemented Loom, an application that generates synthetic time series data by using mathematical models and given time series.

Pesch et al. [54] proposed an innovative methodology for synthetic wind power time series generation based on Markov-chain statistical model.

Schaffner et al. [62] proposed two approaches for the simulation of traffic rate generated by tenants sending requests to a server cluster.

Kang et al. [41] presented an innovative technique for efficient time series generation, based on Gaussian mixture autoregressive (MAR) models for non-Gaussian and nonlinear time series simulation. This approach has been implemented in a shiny application for time series generation [12].

Bagnall et al. [4] implemented a simulator that generates time series data from different shape settings for time series classification algotrithms evaluation purpose.

Vinod et al. [76] generated ensembles for time series data using a maximum entropy bootstrap technique. This approach allowed to preserve multiple features such as shape of data and peaks of the original data. This make them suitable for statistical inference.

4 DATA COLLECTION AND REALISTIC TIME SERIES GENERATION

This chapter focuses mainly on the data set creation and on the techniques adopted to generate realistic time series data. The first part gives an introduction to the data collection process and describes the metrics collected. The second part describes the time series generation and the generation of anomalies for the dataset creation.

4.1 Data Collection

Time-series data were collected from the services by querying Prometheus server using its HTTP API. The data collected cover four weeks of time and time series data have steps of 60 seconds.

A total of 168 metrics were collected from the different tools which include Apache Kafka (120 metrics collected from Kafka brokers, producers, and consumers), Apache ZooKeeper (22 metrics collected from ZooKeeper nodes), Java Virtual Machine (16 metrics from threads, classes, and memory of JVMs), Process (6 standard metrics from processes) and Java Management Extension (4 metrics from JMX configurations).

Each of these metrics is exposed by multiple instances (microservices). All the metrics have been collected, merged and aligned in time in a table which has a column for each KPI exposed by each instance. After the collection process, it was discovered that many instances were not producing relevant data because they were idle. Therefore, data were cleaned by removing constant metrics and null values.

The resulting table had 168 KPIs exposed by 25 instances for a total of 4200 columns plus the timestamp. An example of the table is illustrated in Table **??**.

Time	Variables (168)																			
	Kafka							ZooKeeper							Others					
	M_K1			...	M_K120			M_Z1			...	M_Z22			MO1			...	MO26		
	I1	...	I25	...	I1	...	I25	I1	...	I25	...	I1	...	I25	I1	...	I25	...	I1	...	I25
01/01/19 11:00:00	3	...	4	...	1	...	2	7	...	5	...	2	...	3	1	...	9	...	7	...	8
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
07/01/19 11:00:00	4	...	8	...	1	...	6	8	...	4	...	9	...	6	6	...	9	...	1	...	9

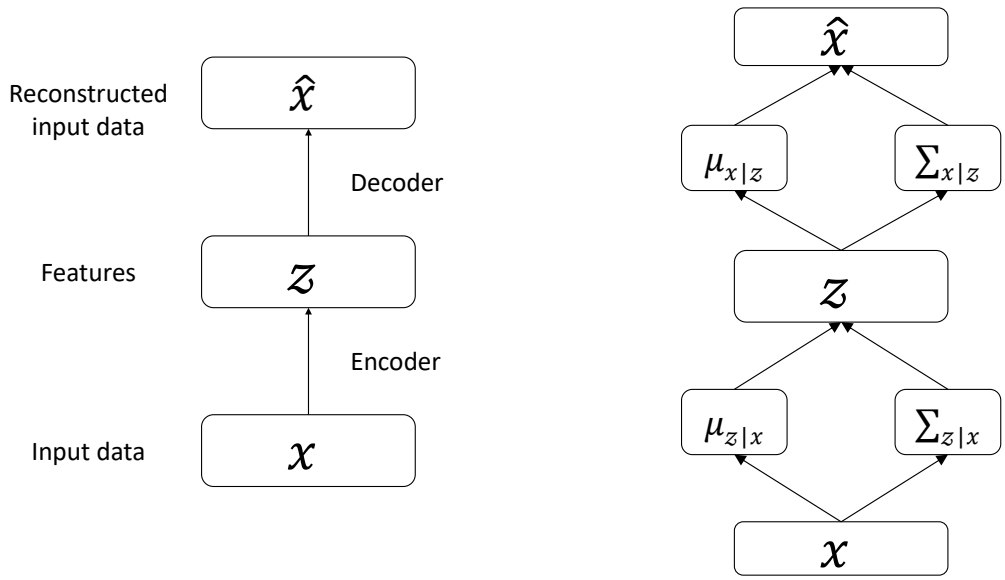
Table 4.1. *The data collected weekly*

4.2 Realistic time series generation

The study researched the availability of data sets with labeled time series regarding cloud-native systems' metrics. The research concluded that this kind of data are difficult to find because they can only be produced in large environments and companies are not willing to share them for privacy concerns. Moreover, the unavailability of labeled data is one of the greatest challenges when applying machine learning because data labeling is not automatic and therefore usually very expensive. In the field of anomaly detection, one of the most common ways of generating an anomaly is by picking from the distribution D a set of anomaly features F such as the magnitude m and the duration d of the anomaly. This way is straightforward but it restricts the types of anomalies that can be created, leads to the problem of over-fitting the models to the synthetic training set, and usually performs poorly on the test set because the synthetic and test set are rarely similar. Recently, an innovative method has been proposed at Facebook[47] that consists of leveraging the features of Variational autoencoder to generate realistic time-series wih outliers at predefined points.

4.2.1 Variational Autoencoder

Variational autoencoder is the generative counterpart of the deterministic autoencoder. As shown in Figure 4.1, it has the same architecture that is based on two models, an encoder and a decoder, but it applies a probabilistic interpretation to them.



***(a)** Compact view* ***(b)** Expanded view*

***Figure 4.1.** Autoencoder architecture*

It assumes that the data set $\{x^{(i)}\}_{i=1}^{N}$ is composed of N i.i.d. samples of some variable

x. Moreover, it assumes that data are generated by a random process with continuous latent variable z and x is generated by the conditional distribution $p_\theta(x|z)$, where p_θ is a probability distribution with parameters θ. This provides a probabilistic interpretation of the decoder network, that given a latent variable z, generates a a sample x in the data space.

The role of the encoder is to take a sample x from data space and generate z, a latent sample from the posterior density distribution $p_\theta(z|x)$.

The training objective of Variational autoencoders is a tractable lower bound to the log-likelihood:

$$\log p_\theta(x) \geq E_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] = -(x) \tag{4.1}$$

$$(x) = D_{KL}(q_\phi(z|x)||p_\theta(z)) - E_{q_\phi(z|x)}[\log p_\theta(x|z)] \tag{4.2}$$

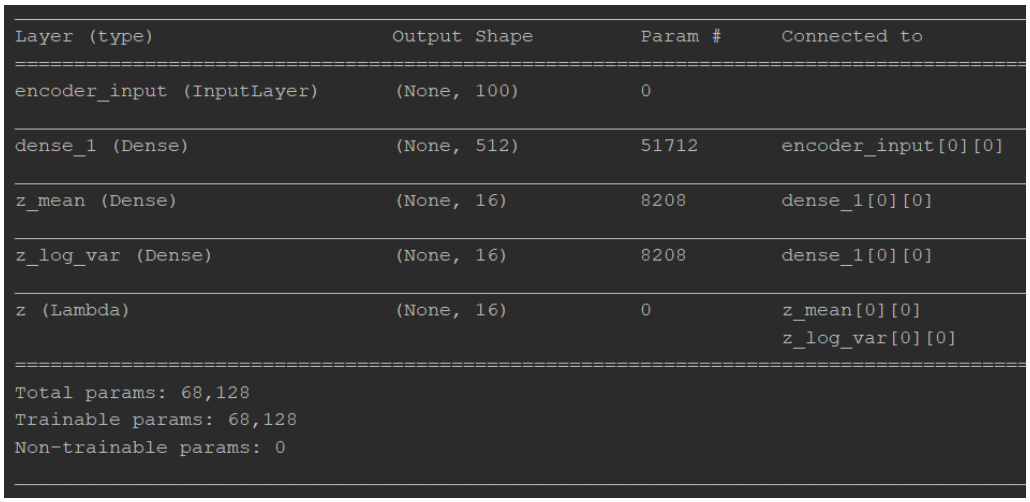
Where:

- D_{KL} is the Kullback-Leibler divergence.
- $E_{q_\phi(z|x)}[\log p_\theta(x|z)]$ is the reconstruction error and represents that likelihood that the input data would be reconstructed by the model.
- $D_{KL}(q_\phi(z|x)||p_\theta(z))$ is the Variational regularization term and represents the KL-divergence between the encoder-induced latent distribution and the true prior on the latent distribution. This term encourages the approximate posterior $q_\phi(z|x)$ to be close to $p_\theta(z)$.

In Variational auto-encoder, z is sampled from a normal distribution parametrized by the mean and the variance. After training the model, new time series could be generate by sampling from latent space z.

4.2.2 Implementation

The varational autoencoder was implemented in the Python language using Keras [15]. It is composed of two elements, the Encoder and the Decoder. The encoder maps sequences of 100 data points into points in the latent space and the details about its neural network architecture layers can be seen in Figure 4.2.

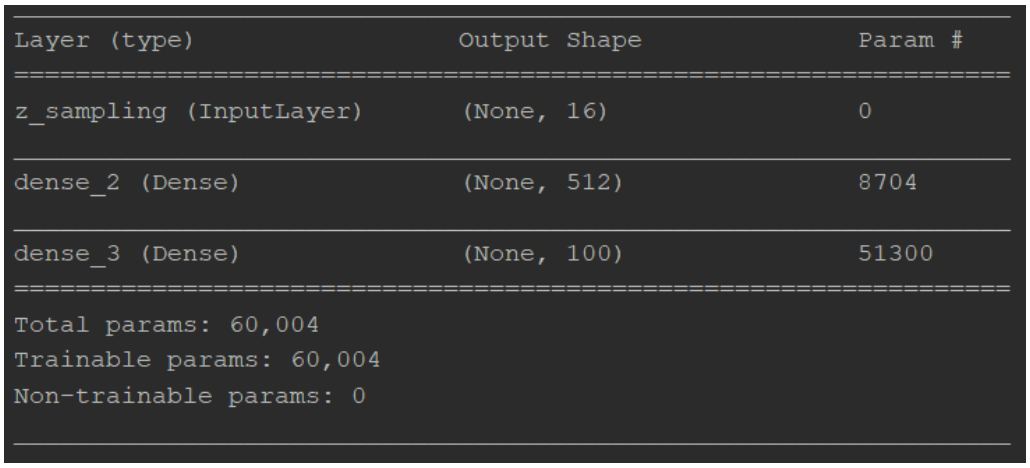


Layer (type)	Output Shape	Param #	Connected to
encoder_input (InputLayer)	(None, 100)	0	
dense_1 (Dense)	(None, 512)	51712	encoder_input[0][0]
z_mean (Dense)	(None, 16)	8208	dense_1[0][0]
z_log_var (Dense)	(None, 16)	8208	dense_1[0][0]
z (Lambda)	(None, 16)	0	z_mean[0][0] z_log_var[0][0]

Total params: 68,128
Trainable params: 68,128
Non-trainable params: 0

Figure 4.2. *Encoder implementation*

The decoder, on the other hand, decodes the samples taken from the latent space into sequences of 100 data points and the details about its neural network architecture layers in Figure 4.3.



Layer (type)	Output Shape	Param #
z_sampling (InputLayer)	(None, 16)	0
dense_2 (Dense)	(None, 512)	8704
dense_3 (Dense)	(None, 100)	51300

Total params: 60,004
Trainable params: 60,004
Non-trainable params: 0

Figure 4.3. *Decoder implementation*

For each KPI, first the autoencoder is trained using the time series data of the respective KPI as input. This process generates the latent space, a space where each sample is encoded by forming a multidimensional Gaussian distribution. The latent space created has 16 dimensions. A two-dimensional representation of the latent space can be seen in Figure 4.4.

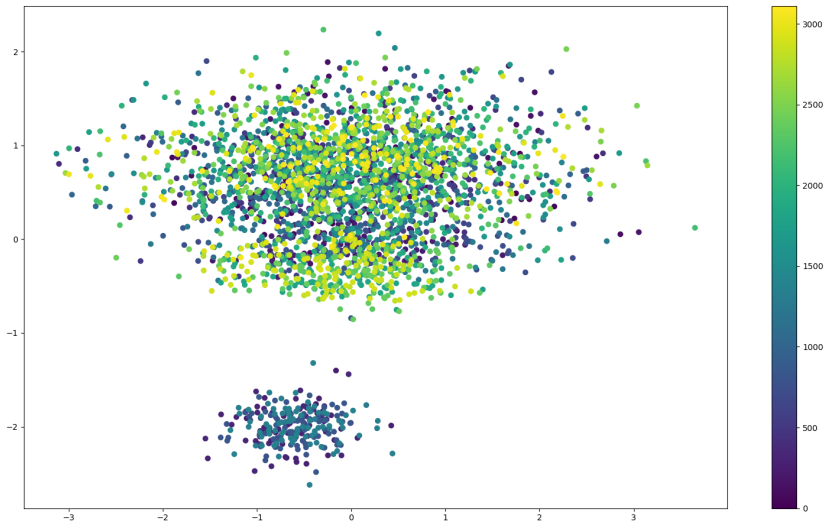
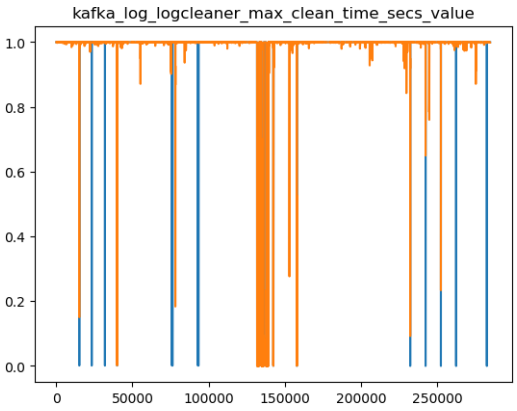


Figure 4.4. *2D representation of the latent space*

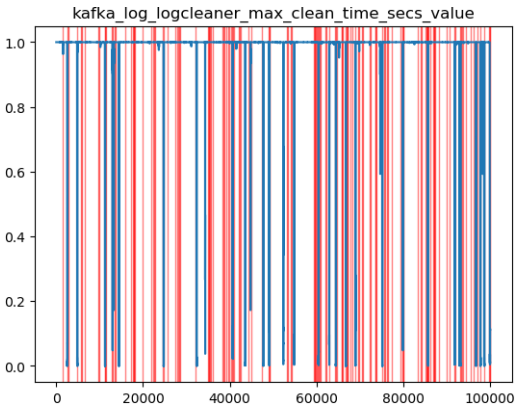
The algorithm operates by sampling the latent space z randomly from the normal and abnormal distribution. Since the percentage of anomalies was set to 10%,the latent space has been sampled from the outlier region 10% of the times, creating anomalies at deterministic locations. The 90% of the times, the latent space z was sampled from the normal distribution.

4.2.3 Results

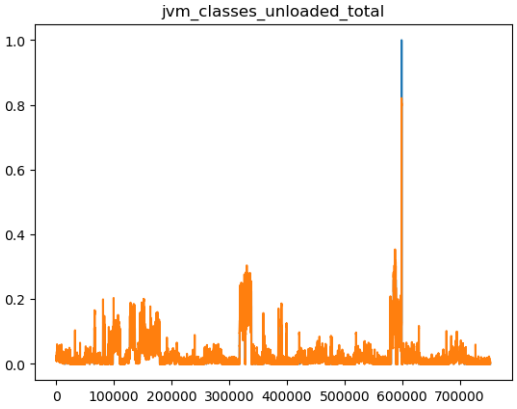
The result is a dataset with 168 KPIs having 100k data points and labeled having 10% of anomalies (10k data points). Figure 4.5 shows some examples of the original time series and the time series generated, with 10% of anomalies. The figures on the left show the original time series in blue, and the respective prediction made by the autoencoder in orange, to be sure the autoencoder has learned accurately. The figures on the right show the generated time series in blue, with the highlighted anomalies in red.



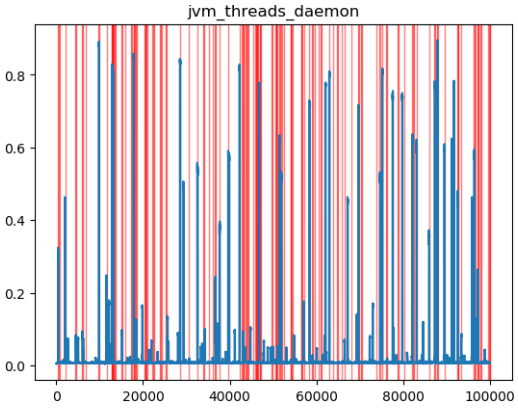
(a) *Example 1: original*



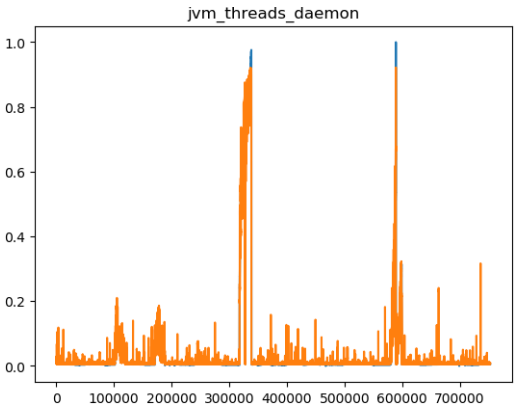
(b) *Example 1: generated*



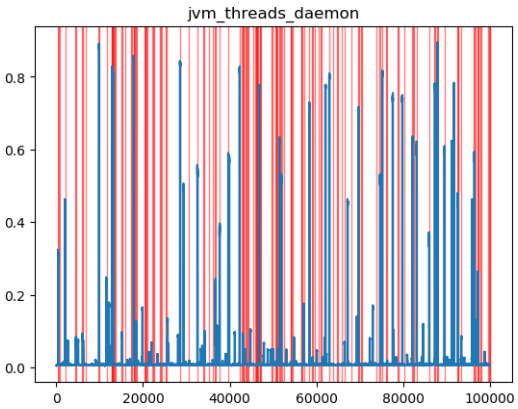
(c) *Example 2: original*



(d) *Example 2: generated*



(e) *Example 3: original*



(f) *Example 3: generated*

Figure 4.5. *Autoencoder training output and generated time series with 10% of anomalies*

5 CLASSIFICATION-BASED ANOMALY DETECTION

This chapter presents the empirical study which is a case study based on the guidelines defined by Runeson and Höst [59]. The objective of this study has been formulated by using the Goal/Question/Metric (GQM) template [5].

With respect to the quality attributes accuracy, training and testing time, the following questions were derived:

Accuracy-related Questions

Q1 Is there a Machine Learning algorithm that can accurately detect performance-related anomalies in cloud-native systems?

Q1.1 Which Machine Learning algorithm has higher accuracy in detecting performance-related anomalies in cloud-native systems?

Training- and Testing-Time-related Questions

Q2 Which Machine Learning algorithm can accurately detect performance-related anomalies with the shortest training time?

Metrics Importance-related Questions

Q3 What are the most important metrics to be considered when detecting performance-related anomalies?

Q4 How many components are necessary to accurately detect performance-related anomalies?

In order to answer to these questions, a set of metrics that are symptoms of performance anomalies need to be identified. For this purpose, six KPIs have been identified. These KPIs are considered to be fundamental to the system and their thresholds should not be exceeded (see Table 5.1).

Table 5.1. *Dependent Variables (KPIs)*

Metric	**Description**	**Threshold**
Min Fetch Rate	The minimum rate at which the consumer sends fetch requests to the broker. If a consumer is dead, this value drops to roughly 0.	> 0.5
% Network Processor Idling Time	Average fraction of time the network processor threads are idle. The values are between 0 (all resources are used) and 1 (all resources are available).	> 0.3
Max Message Lag	Maximum lag in messages between the follower and leader replicas.	< 50
Avg Request Latency	Amount of time it takes for the server to respond to a client request (since the server was started).	< 100
Request Queue Size	Number of requests queued in the server. Goes up when the server receives more requests than it can process.	< 10
# Pending Sync	The number of pending syncs from the followers.	< 10

Each question will be further explained in the following sub-sections.

5.1 Accuracy

To assess the detection accuracy of the different Machine Learning algorithms, we performed a 10-fold cross-validation dividing the data into ten parts. In other words, we trained the models ten times, always using 1/10 of the data as a testing fold. The data were split into ten sequential parts, thus respecting the temporal order and the proportion of data for each project. The models were trained iteratively on groups of data preceding the test set. Furthermore, the temporal order was also respected for the groups included in the training set: For example, in fold 1 we used group 1 for training and group 2 for testing. In fold 2 groups 1 and 2 were used for training and group 3 for testing, and so forth for the remaining folds.

As accuracy metrics, precision and recall were calculated at first. However, as suggested by [55], these two measures present some biases as they are mainly focused on positive examples and predictions, and therefore do not capture any information about the rates and kinds of errors made.

The contingency matrix (also called confusion matrix) and the related f-measure help to

overcome this issue. Moreover, as recommended by Powers [55], the Matthews Correlation Coefficient (MCC) should also be considered to understand any potential disagreement between the actual values and the predictions, as it involves all four quadrants of the contingency matrix. From the contingency matrix, were retrieved multiple measures which include the *true negative rate* (TNR), the *false positive rate* (FPR) and the *false negative rate* (FNR). The TNR measures the percentage of negative samples correctly categorized as negative. The FPR measures the percentage of negative samples misclassified as positive. The FNR measures the percentage of positive samples misclassified as negative. The *true positive rate* (TPR) measure was left out as it is equivalent to the recall.

The way these measures were calculated can be found in Table 5.2.

Table 5.2. *Accuracy Metrics Formulas*

Accuracy Measure	**Formula**
Precision	$\frac{TP}{FP+TP}$
Recall	$\frac{TP}{FN+TP}$
MCC	$\frac{TP*TN-FP*FN}{\sqrt{(FP+TP)(FN+TP)(FP+TN)(FN+TN)}}$
f-measure	$2*\frac{precision*recall}{precision+recall}$
TNR	$\frac{TN}{FP+TN}$
FPR	$\frac{FP}{TN+FP}$
FNR	$\frac{FN}{FN+TP}$

TP: True Positive; TN: True Negative; FP: False Positive; FN: False Negative

Finally, the Receiver Operating Characteristics (ROC) and the related Area Under the Receiver Operating Characteristic Curve (AUC) were calculated. The ROC curve represents the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one and has been calculated and plotted using the ”roccurve()” function of the scikit-learn [53] library.They .

5.2 Training and Testing Time

Regarding this aspect, the training and testing time (in seconds) were collected and compared for each algorithm. The aim was to be able to select one algorithm that can be trained with the shortest training time and with a high level of accuracy.

5.2.1 Context

The system is composed of several microservices running on top of Kubernetes and communicating using a lightweight message bus (Apache Kafka). The size of the system requires multiple Kafka brokers and therefore the use of Zookeeper to coordinate the different Kafka instances.Different metrics were collected from Kafka, Zookeeper, and other tools.

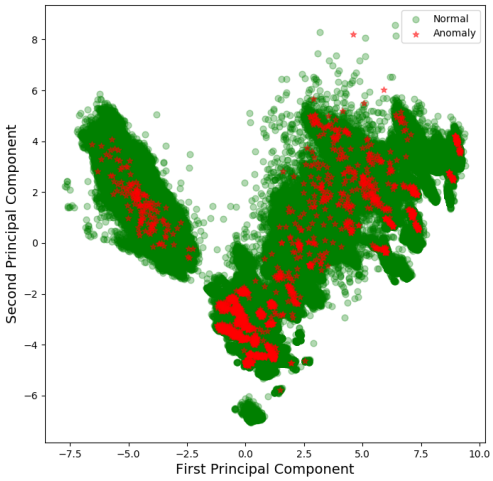
5.3 Data Collection and Preparation

From the dataset created previously, data have been merged grouping by metric name. An example of the resulting table is reported in Table 5.3.

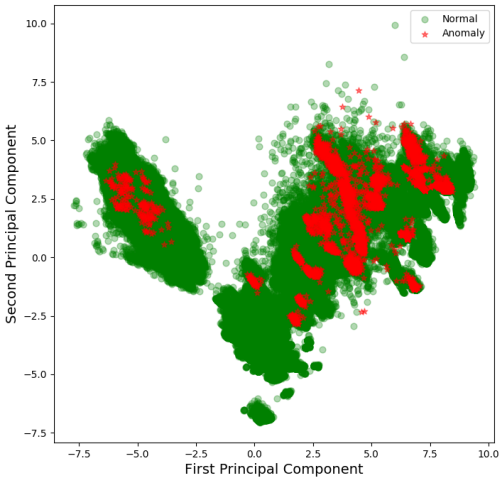
Table 5.3. *The data collected weekly*

	Variables (168)								
	Kafka			**ZooKeeper**			**Others**		
Time stamp	MK1	...	MK120	MZ1	...	MZ22	MO1	...	MO26
01/01/19-11:00:00	2.00	...	6.34	8.56	...	1.27	3.87	...	2.01
...	...	...	...	...	...	...	...	...	...
07/01/19-11:00:00	5.11	...	8.00	4.33	...	3.04	6.72	...	9.20

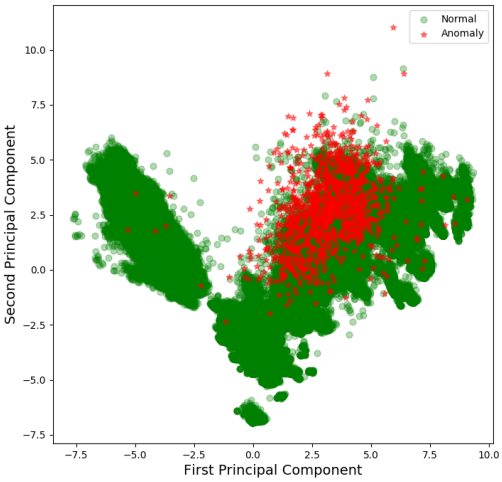
Data have been labeled considering anomaly values for the six metrics exceeding the default thresholds (Table 5.1). Unexpectedly, one of the dependent variables (# Pending Sync) never exceeded the threshold in the monitoring time frame. Therefore, we only considered the remaining five dependent variables for the analysis. Figure 5.1 shows a graphycal representation of the data. The green points are normal values, instead the red points are anomalous values.



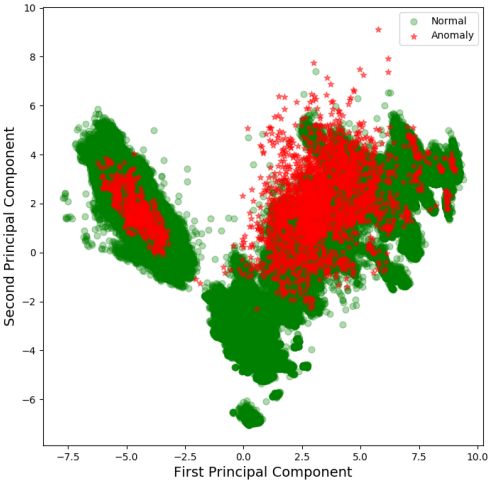
(a) *Request Queue Size*



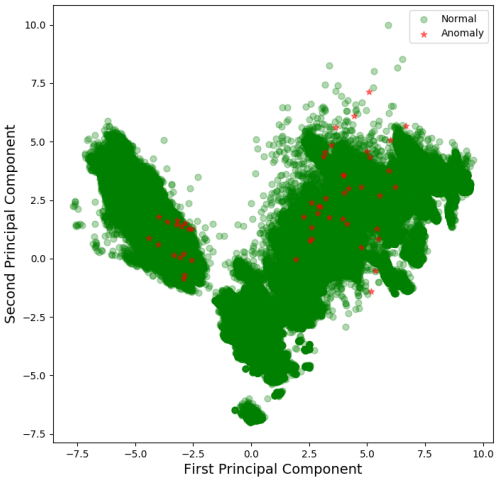
(b) *Avg Request Latency*



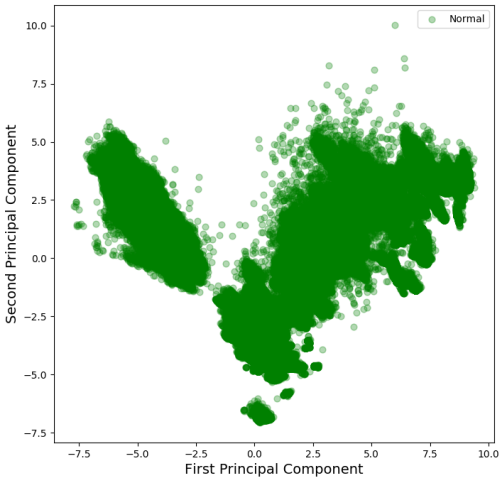
(c) *Min Fetch Rate*



(d) *Max Message Lag*



(e) *% Network Processor Idling Time*



(f) *# Pending Sync*

Figure 5.1. *2D representation of anomalies*

The result of this process consisted of five tables, which were used to train and test the Machine Learning algorithms. Table 5.4 shows an example of the data represented in one of the five csv files.

Table 5.4. *Example of Labeled Data*

	Dependent Variable	**Independent Variables**			
Time stamp	DM1	M1	M2	...	M167
01/01/19-11:00:00	1	1.27	3.87	...	2.01
...	...	...	...	...	...
07/01/19-11:00:00	0	4.57	3.86	...	2.90

5.4 Data Analysis

This phase consists of multiple analyses performed on the data set in order to answer the research questions formulated earlier. In this phase the eight algorithms described in Section 2.3 have been applied to each of the five data sets in order to verify whether there are dependencies between each dependent variable based on the independent ones. During this process the accuracy, training time and test time were collected from each algorithm, so that it was possible to compare their accuracy by means of the accuracy measures reported in Section 5.1 in addition to their training and test time. In order to answer to the third research question (which are the metrics that contribute more to the prediction of anomalies) two different methods were applied: one statistical method the Principal Component Analysis (PCA) and one regarding Machine learning, the drop-column algorithm.

Principal component analysis is a statistical algorithm that reduces data dimension while retaining most of the information by creating new components that summarize the data. It is widely used in data mining for datasets investigation. In PCA, new orthogonal components (latent variables or principal components) are obtained by maximising the data variance. The total of the principal components (factors) is usually much lower than the total of original variables, so that the data can be visualised in a low-dimensional space. While principal components analysis decreases the space dimension, it does not decrease the number of the original components, as it uses all of them for the generation of the new latent variables (principal components). This aspect of PCA is leveraged to figure out the importance of features. In fact, features with the highest contribution to these components are the most important.

Drop-column mechanism [52] is a simplified alternative of the exhaustive search technique [78], which iteratively tests every subset of variables for their classication performance. The full exhaustive search is very time-consuming, because it requires 2X train-evaluation steps, where X is the dimension of the feature space. Instead, in the drop-

column technique, individual features are dropped one at a time, instead of all possible groups of features. This means that a model is trained X times for a X-dimensional feature space, iteratively removing one feature at a time, from the first to the last of the data set. The difference in cross-validated test accuracy between the newly trained model and the baseline model (the one trained with the full set of features) defines the importance of that specific feature. The more the accuracy of the model drops, the more important is the specific feature for the classication. The importance of the metrics was not calculated for all the machine learning models described, but only for the most accurate model (cross-validated with all X features), because the feature importances of a classier with lower accuracy performance were likely to be less reliable.

6 RESULTS

The data extracted from four weeks reported more than 800k rows, resulting in a 700MB csv file. Since the data were collected from a real industrial system, they cannot be shared in this thesis.

The eight Machine Learning algorithms were executed using a Linux Ubuntu machine with 24 cores and 90GB RAM.

In the next sub-sections, the results obtained after analyzing the collected data are presented.

Q1: Is there a Machine Learning algorithm that can accurately detect performance-related anomalies in cloud-native systems?

All the accuracy measures adopted reported consistent results.

Only three variables can be predicted with an accuracy (AUC) higher than 90%, while two variables can be predicted with an accuracy (AUC) higher than 80%. The comparison of the accuracy of the different Machine Learning models revealed that XGBoost is the most accurate model for four out of five KPIs, while in one case, ExtraTrees performed better than the others. Below the accuracy measures for each metric are reported in addition to charts representing the AUC comparison for each model.

Min Fetch Rate

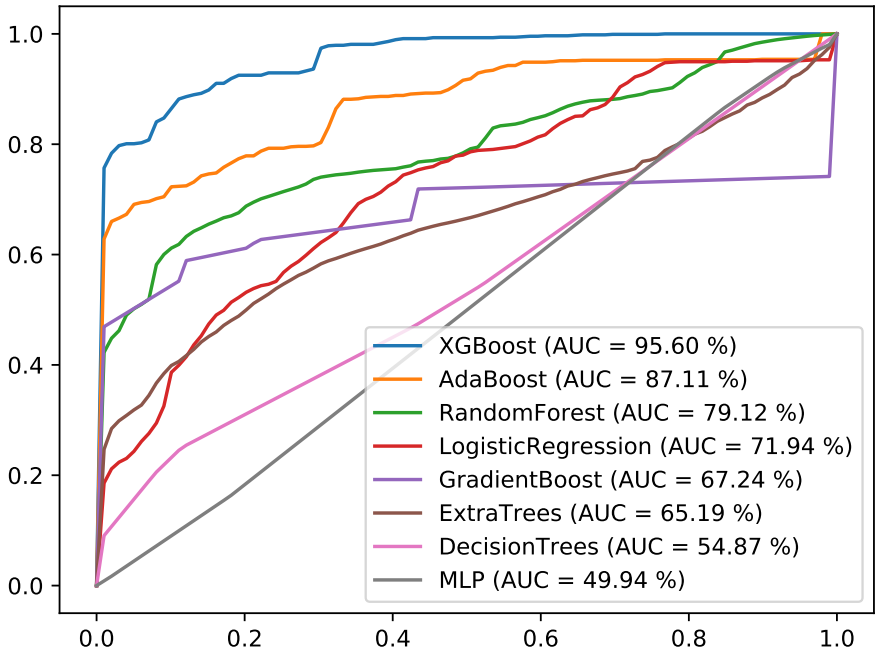


Figure 6.1. *Comparison of algorithms' accuracy for Min Fetch Rate KPI*

Classifier	TNR	FNR	FPR	Recall	Prec	f-meas	MCC
AdaBoost	95.14	77.91	4.86	22.09	66.78	23.20	30.85
DecisionTrees	88.49	78.67	11.52	21.32	34.95	11.45	15.70
ExtraTrees	89.51	92.16	10.49	7.83	54.33	4.01	9.83
GradientBoost	90.31	57.94	9.69	42.06	55.84	31.51	35.44
LogisticRegression	87.76	81.25	12.24	18.75	26.67	7.21	9.93
MLP	98.00	99.91	2.00	0.09	0.01	0.01	-0.21
RandomForest	89.85	91.14	10.14	8.86	65.06	5.90	13.15
XGBoost	97.38	81.41	2.61	18.58	64.37	17.39	24.99

Table 6.1. *Accuracy metrics for Min Fetch Rate KPI*

% Network Processor Idling Time

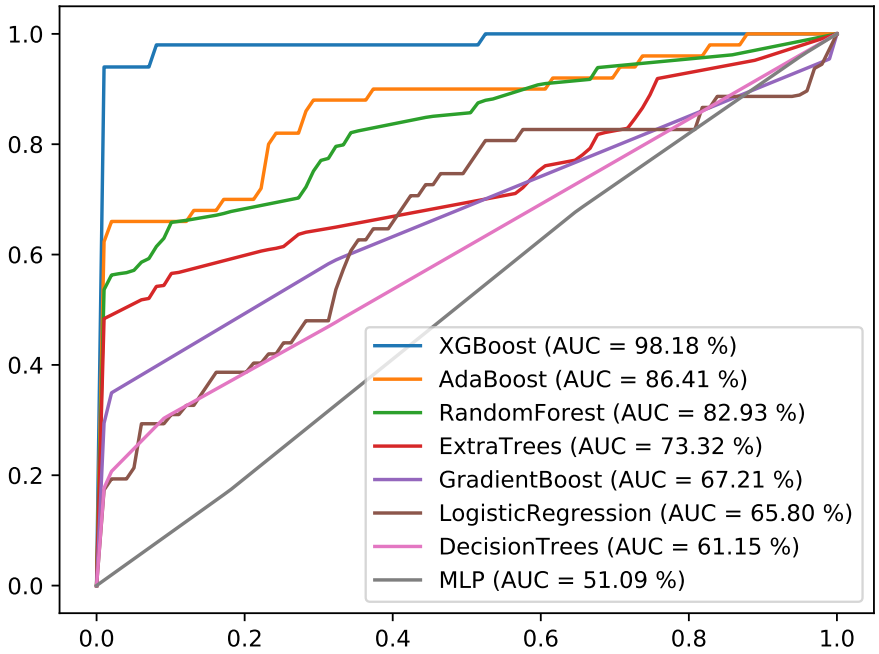


Figure 6.2. *Comparison of algorithms' accuracy for % Network Processor Idling Time KPI*

Classifier	TNR	FNR	FPR	Recall	Prec	f-meas	MCC
AdaBoost	96.16	82.67	3.842	17.33	11.95	10.60	11.84
DecisionTrees	92.43	70	7.566	30	22.71	13.86	15.79
ExtraTrees	99.05	96	0.95	4	1.18	1.82	2.14
GradientBoost	93.35	56.67	6.65	43.33	32.80	16.30	21.64
LogisticRegression	87.56	88	12.44	12	0	0.01	-0.01
MLP	98.20	100	1.80	0	0	0	-0.04
RandomForest	98.79	91.33	1.21	8.67	4.12	3.90	4.84
XGBoost	99.96	83.33	0.03	16.67	23.15	10.94	14.41

Table 6.2. *Accuracy metrics for % Network Processor Idling Time KPI*

Request Queue Size

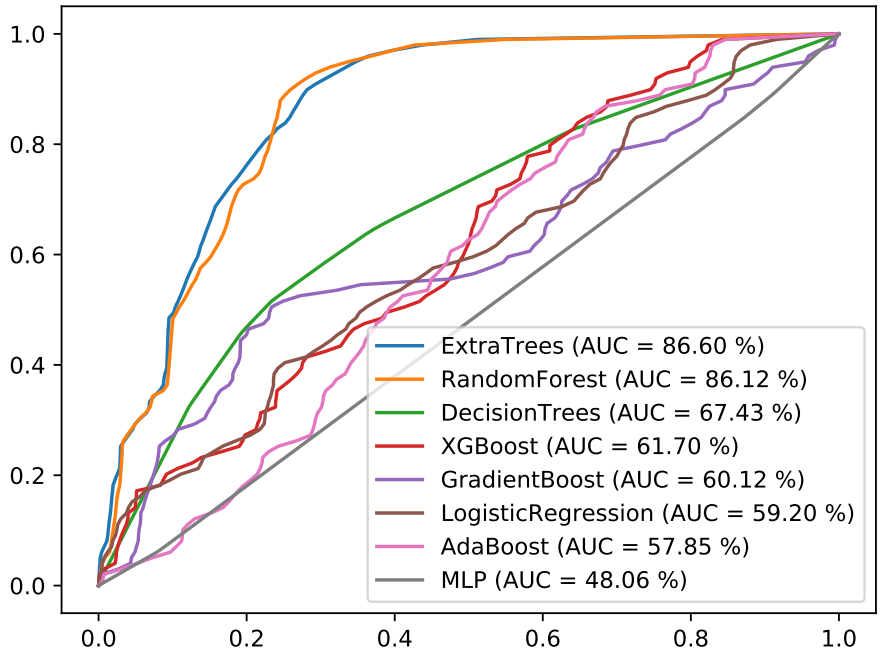


Figure 6.3. *Comparison of algorithms' accuracy for Request Queue Size KPI*

Classifier	TNR	FNR	FPR	Recall	Prec	f-meas	MCC
AdaBoost	90.09	85.31	9.91	14.69	14.55	28.11	0.59
DecisionTrees	84.37	49.50	15.63	50.50	7.29	14.76	3.93
ExtraTrees	91.77	54.48	8.23	45.51	5.93	10.93	3.98
GradientBoost	82.10	50.20	17.90	49.80	7.34	14.01	3.35
LogisticRegression	96.79	84.91	3.21	15.09	5.20	9.47	1.42
MLP	97.67	99.38	2.32	0.62	10.88	18.25	-0.36
RandomForest	91.72	69.32	8.28	30.68	5.91	10.09	2.47
XGBoost	98.05	86.89	1.95	13.11	3.51	6.99	1.33

Table 6.3. *Accuracy metrics for Request Queue Size KPI*

Avg Request Latency

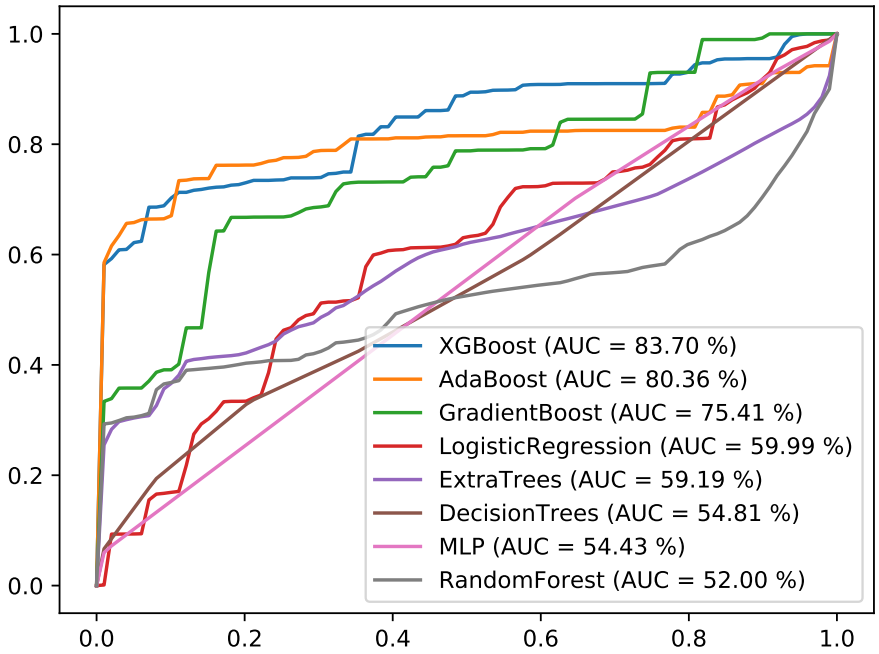


Figure 6.4. *Comparison of algorithms' accuracy for Avg Request Latency KPI*

Classifier	TNR	FNR	FPR	Recall	Prec	f-meas	MCC
AdaBoost	94.00	85.30	5.99	14.70	25.49	10.97	12.01
DecisionTrees	78.85	69.18	21.14	30.81	41.37	10.87	15.31
ExtraTrees	85.27	65.81	14.73	34.19	41.28	12.33	18.02
GradientBoost	80.33	79.86	19.67	20.14	21.26	2.97	3.37
LogisticRegression	79.42	96.99	20.58	3.01	0.18	0.28	-4.20
MLP	99.89	94.78	0.11	5.22	3.28	4.03	4.05
RandomForest	92.28	65.54	7.72	34.46	51.96	24.27	29.63
XGBoost	93.92	85.88	6.08	14.11	22.83	10.88	11.16

Table 6.4. *Accuracy metrics for Avg Request Latency KPI*

Max Message Lag

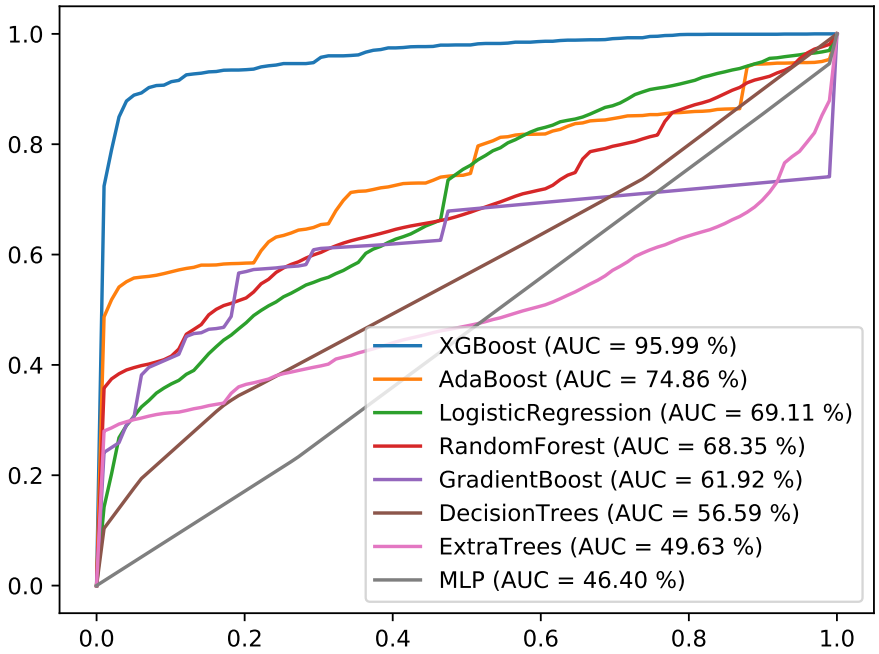


Figure 6.5. *Comparison of algorithms' accuracy for Max Message Lag KPI*

Classifier	TNR	FNR	FPR	Recall	Prec	f-meas	MCC
AdaBoost	94.25	69.38	5.75	30.62	59.36	28.72	33.75
DecisionTrees	80.36	67.11	19.64	32.89	31.88	11.52	14.98
ExtraTrees	76.55	84.47	23.45	15.53	40.18	5.95	8.39
GradientBoost	88.37	64.87	11.63	35.12	37.49	19.62	23.50
LogisticRegression	85.20	70.91	14.80	29.09	14.42	5.43	7.55
MLP	97.30	100	2.70	0	0	0	-0.37
RandomForest	80.14	71.93	19.86	28.07	40.45	11.42	14.32
XGBoost	99.23	57.17	0.77	42.82	67.36	41.30	46.73

Table 6.5. *Accuracy metrics for Max Message Lag KPI*

Q2: Which Machine Learning algorithm can accurately detect performance-related anomalies with the shortest training time?

The training time of all the techniques except Logistic Regression was very short. For the other techniques, in some cases, some sub-optimal technique required a shorter training time than the optimal technique. For example, the testing time of Random Forest was much shorter than the one spent for training XGBoost. However, the difference is in the range of a few minutes. Since the plan is to train the system once a week, the time differences can be considered negligible. The comparisons of the training time, testing time, and AUC for each metric are reported in the tables below.

Min Fetch Rate

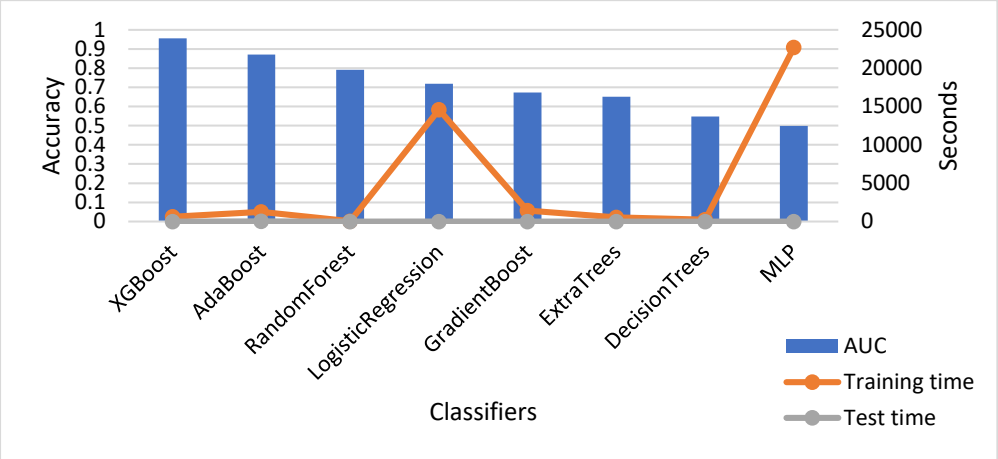


Figure 6.6. *Algorithms accuracy, training and testing time comparison for Min Fetch Rate KPI*

% Network Processor Idling Time

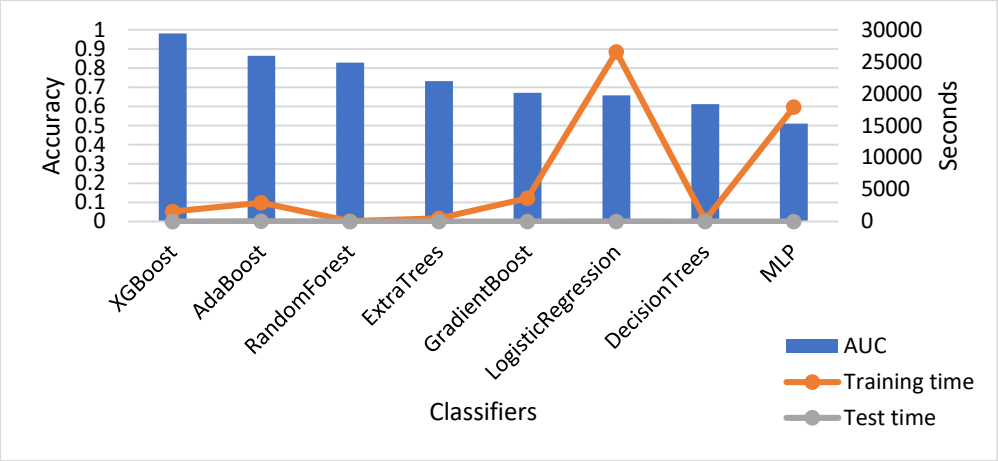


Figure 6.7. *Algorithms accuracy, training and testing time comparison for % Network Processor Idling Time KPI*

Request Queue Size

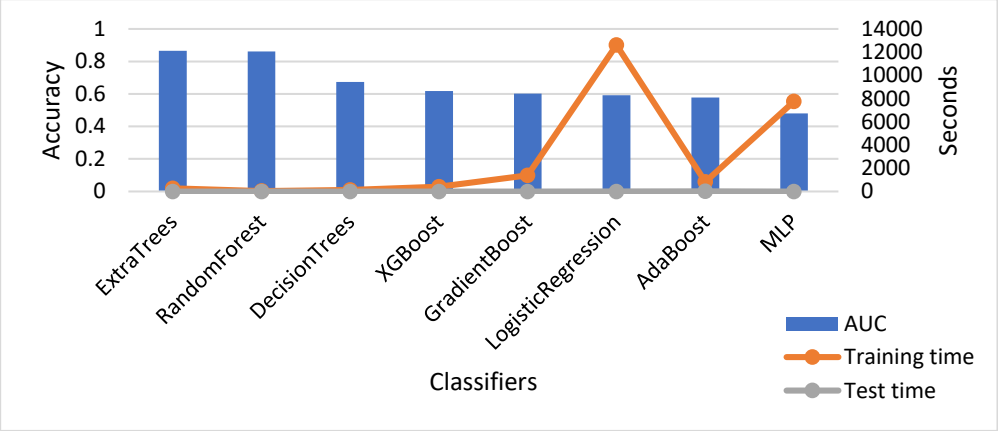


Figure 6.8. *Algorithms accuracy, training and testing time comparison for Request Queue Size KPI*

Avg Request Latency

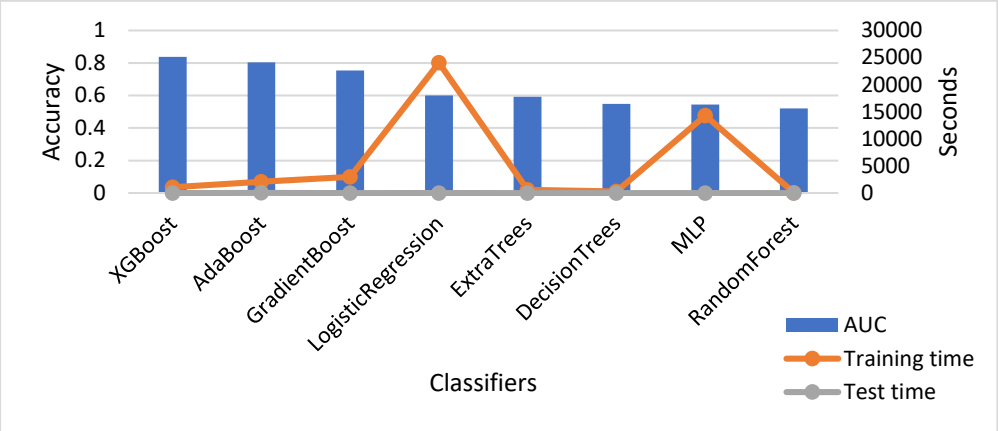


Figure 6.9. *Algorithms accuracy, training and testing time comparison for Avg Request Latency KPI*

Max Message Lag

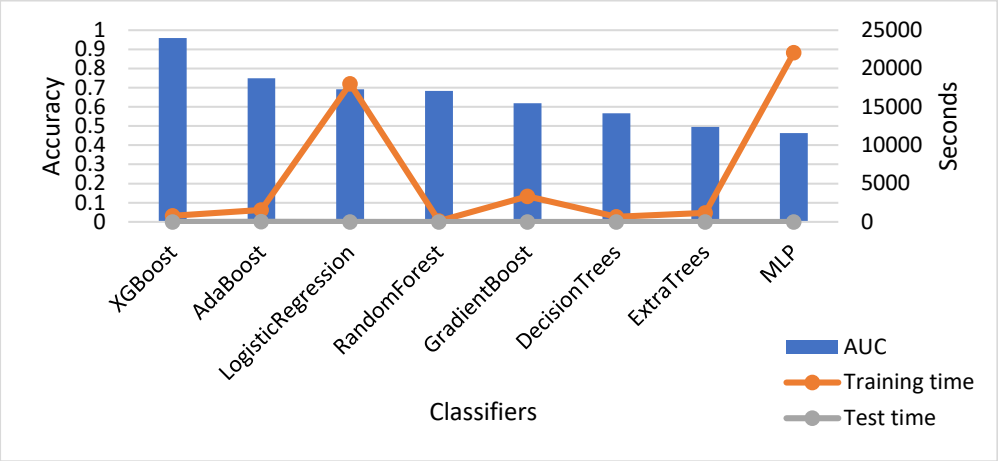


Figure 6.10. *Algorithms accuracy, training and testing time comparison for Max Message Lag KPI*

Q3: What are the most important metrics to be considered when detecting performance-related anomalies?

The two methods adopted, Principal Component Analysis and drop-column mechanism, reported different results. This can be expected due to their different approaches. Below, the first ten most important metrics are reported for both the techniques. The blue lines represent the importance of each feature for the prediction of the respective KPI.

Min Fetch Rate

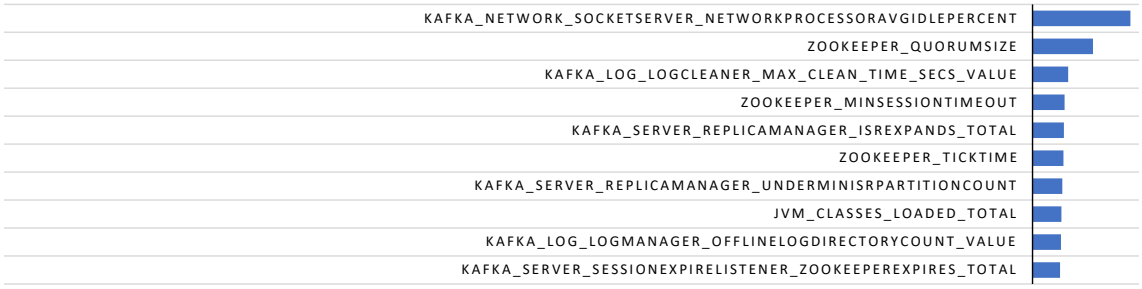


Figure 6.11. *Drop Column algorithm results for Min Fetch Rate KPI*

Figure 6.12. *Principal Component Analysis results for Min Fetch Rate KPI*

% Network Processor Idling Time

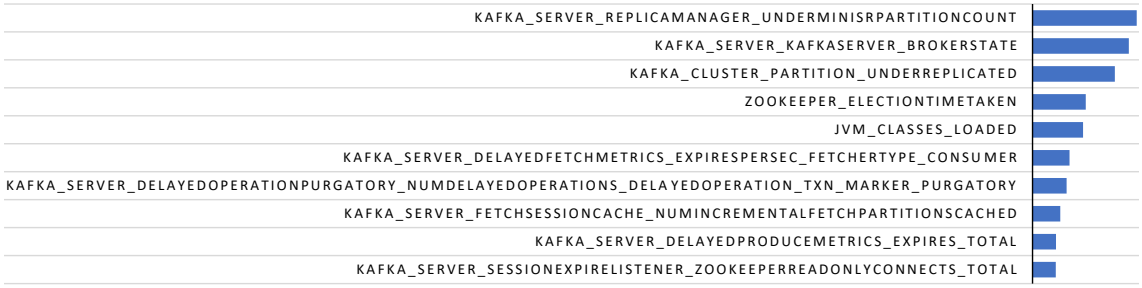


Figure 6.13. *Drop Column algorithm results for % Network Processor Idling Time KPI*

Figure 6.14. *Principal Component Analysis results for % Network Processor Idling Time KPI*

Request Queue Size

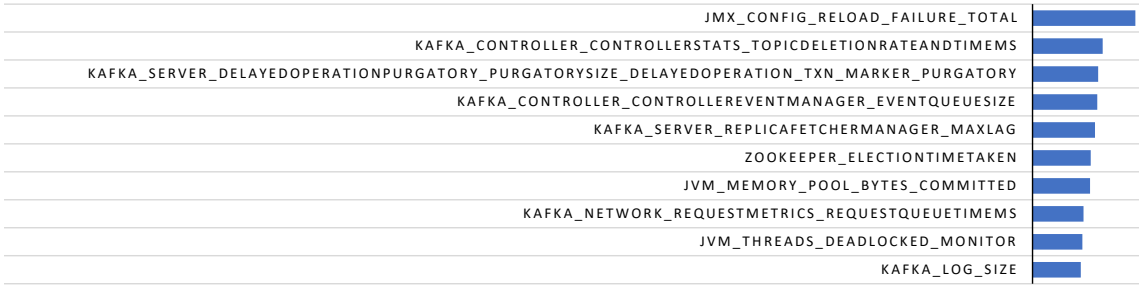


Figure 6.15. *Drop Column algorithm results for Request Queue Size KPI*

Figure 6.16. *Principal Component Analysis results for Request Queue Size KPI*

Avg Request Latency

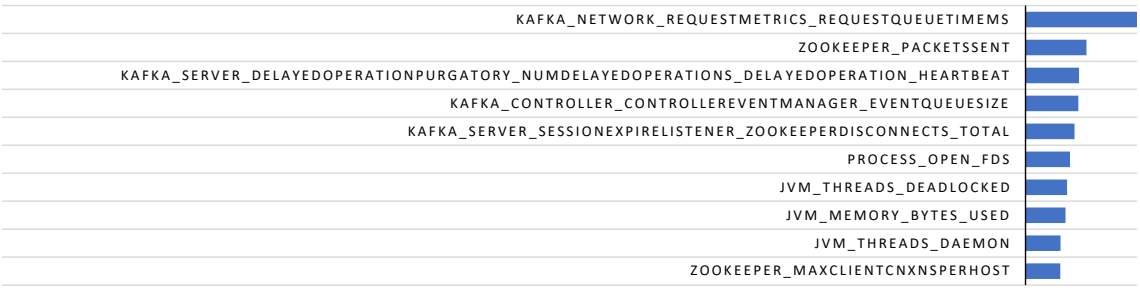


Figure 6.17. *Drop Column algorithm results for Avg Request Latency KPI*

Figure 6.18. *Principal Component Analysis results for Avg Request Latency KPI*

Max Message Lag

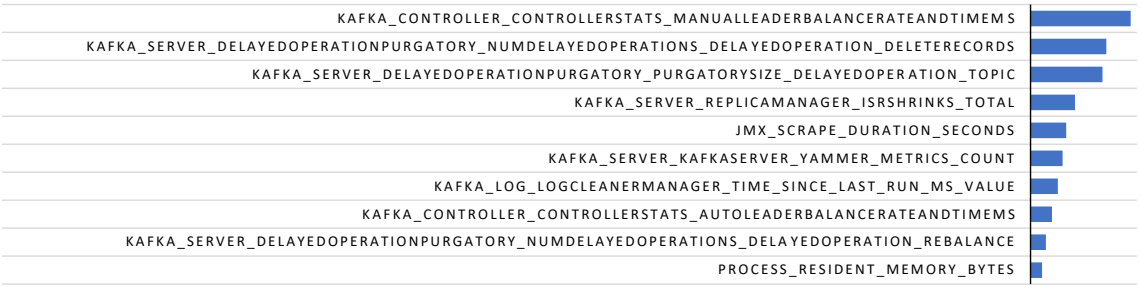


Figure 6.19. *Drop Column algorithm results for Max Message Lag KPI*

Figure 6.20. *Principal Component Analysis results for Max Message Lag KPI*

Q4: How many components are necessary to accurately detect performance-related anomalies?

By performing the Principal Component Analysis for dimensional reduction on the data, it was possible to see how many components are necessary and how many can be removed for the prediction of the six KPIs.

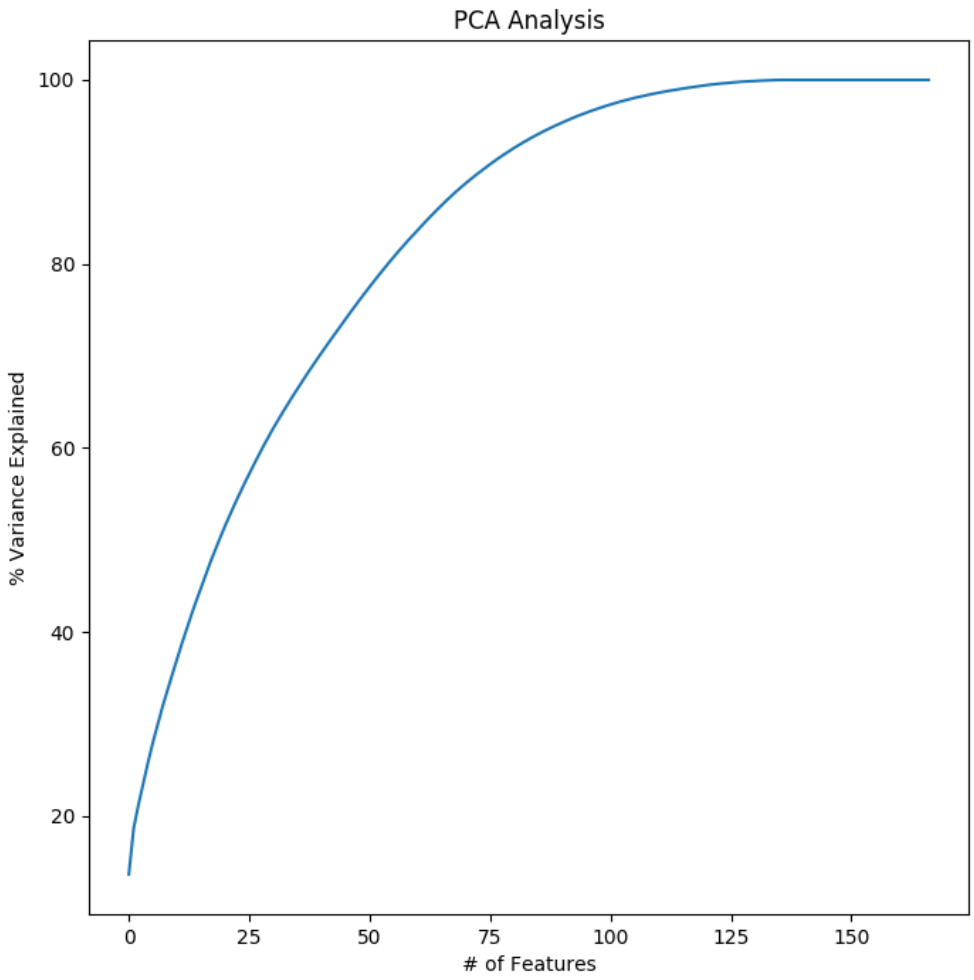


Figure 6.21. *Variance explained by the 168 KPIs*

Since the six charts produced are exactly identical, only one of them is reported in Figure 6.21. Based on these results it is possible to see that 100 components are enough to have around 95% of information. Instead, 120 factors are needed to have almost 100% of information.

7 DISCUSSION

7.1 Discussion

The application of the eight Machine Learning techniques on the data collected at runtime from the cloud-native system showed important dependencies between performance-related KPIs and the metrics collected in the system.

The most important outcome is that performance-related anomalies can be detected by monitoring a limited number of metrics collected at runtime. Some techniques require excessive training time (Logistic Regression). However, other techniques, such as XGBoost, provide a very high level of accuracy in four out of five cases with a brief training time. In the case of the Request Queue Size KPI, ExtraTrees algorithm permormed better than the others.

As for the metrics that can be used to predict the anomalies, the PCA reported the metric *"zookeeper-ElectionType"* as the most important predictor for all the five KPIs. Instead the drop-column algorithm reported that *"Server replica manager - underminisr partition count"* is the best predictor for % of network processor idling time, *"manual leader balance rate and time"* for max message lag, *"% of network processor idling time"* for min fetch rate, *"Request Queue Time"* for Avg request latency, and *"Number of Configuration Reload Failures"* for request queue size.

The final result is that the number of components used for the training can be easily reduced to 120 preserving the same amount of information for the prediction of the anomalies. This result is important because by reducing the number of features, training and testing performances will improve.

The result of this study could help other companies to understand how to monitor cloud-native systems and especially how to detect whether some KPIs they consider relevant are dependent on other metrics they can collect.

7.2 Threats to Validity

This work has several limitations. It is not excluded that different implementations of the Varational Autoencoder could have yielded better results and that a larger amount of data or a longer time span, could have produced different results. There is also the possibility

that other statistical or Machine Learning approaches might have yielded similar or even better accuracy, than modeling approaches adopted in this work. It is not excluded the possibility of other metrics predicting one of these KPIs better, but at the moment, it is not possible to change the configuration of the monitoring systems and to add more metrics.

8 CONCLUSION

8.1 Conclusion

This thesis concentrated on contributing to the improvement of the monitoring system and approaches in cloud native systems by analyzing 168 metrics collected from a huge system built by using the microservices architectural pattern.

The first problem addressed was the lack of publicly available datasets that include metrics collected from large scale projects composed by many components such as orchestrators, load balancers and message buses. For this reason a dataset of realistic time series was created by leveraging an approach based on the Varational Autoencoder. Moreover, by applying a sampling technique proposed at Facebook, it has been possible to generate anomalies at deterministic locations, creating in this way a dataset of labeled time series that could be used to training machine learning models.

The second part of this thesis focused on understanding whether it is possible to predict anomalies from the different services composing the system that can degrade the system performance, so as to take actions before performance decreases significantly or before the system fails.

For this reason an empirical study was performed by comparing the accuracy of eight different machine learning models and their training and testing time. The six most important metrics were used as labels and the rest as features for anomalies classification. The results showed that there are strong correlations between metrics and that it is possible to predict the anomalies in the system with approximately 90% of accuracy. The best performing algorithm was XGBoost, because it had the highest accuracy and the shortest training and testing time in the majority of the cases.

Moreover statistical and machine learning approaches were applied to identify the most important metrics and to identify redundant information in the metrics.

Consequently, the most important metrics were identified and it was disocvered that the 168 metrics can be meged into 120 components without losing any information. This will make the training process much faster and the models less prone to overfitting.

8.2 Future work

On this topic there are multiple opportunities for future improvements. In the future, the amount of data collected and the number of metrics should be increased for improving the completeness of the data set. Regarding the data set creation, different implementations of the varational autoencoder or even completely different approaches can be investigated. Moreover, the dataset of realistic time series must be updated and refactored by adding more KPIs and increasing the time range from one month to one year. The creation of an automatic tool for metric collection and time series generation for continuous improvements can be considered.

Regarding the empirical study on anomaly detection, further work should include the application of different machine learning algorithms and the application of unsupervised techniques for real time anomaly detection. Moreover, suitable techniques for the prediction of the fault-proneness of the different metrics must be investigated and implemented. The final goal will be the development of a tool or a set of plugins for Prometheus for prediction of anomalies in cloud native environments and a set of dashboards similar to Grafana to help the user in the monitoring of the system.

REFERENCES

[1] S. Ahmad, A. Lavin, S. Purdy and Z. Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing* 262 (2017), 134–147.

[2] T. Ahmed, M. Coates and A. Lakhina. Multivariate online anomaly detection using kernel recursive least squares. *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE. 2007, 625–633.

[3] M. Alzantot, S. Chakraborty and M. Srivastava. Sensegen: A deep learning architecture for synthetic sensor data generation. *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2017, 188–193.

[4] A. Bagnall, A. Bostrom, J. Large and J. Lines. Simulated data experiments for time series classification Part 1: accuracy comparison with default settings. *arXiv preprint arXiv:1703.09480* (2017).

[5] V. R. Basili, G. Caldiera and H. D. Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering* (1994).

[6] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz and S. Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349* (2015).

[7] B. Brazil. *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. " O'Reilly Media, Inc.", 2018.

[8] L. Breiman. Random forests. *Machine learning* 45 (2001).

[9] L. Breiman, J. Friedman, C. J. Stone and R. Olshen. *Classification and regression trees Regression trees*. 1984. ISBN: 978-0412048418.

[10] E. Brophy, Z. Wang and T. E. Ward. Quick and Easy Time Series Generation with Established Image-based GANs. *arXiv preprint arXiv:1902.05624* (2019).

[11] V. Chandola, A. Banerjee and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41.3 (2009), 15.

[12] W. Chang, J. Cheng, J. Allaire, Y. Xie, J. McPherson et al. shiny: Web application framework for r, 2015. *R package version* 1.0 (2018), 14.

[13] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. 2016.

[14] X. Chen, C. Lu and K. Pattabiraman. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. *International Symposium on Software Reliability Engineering*. Nov. 2014, 167–177.

[15] F. Chollet et al. *Keras*. 2015.

[16] D. R. Cox. The Regression Analysis of Binary Sequences. *Journal of the Royal Statistical Society. Series B* 20.2 (1958), 215–242. ISSN: 00359246.

[17] D. J. Dean, H. Nguyen and X. Gu. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. *Proceedings of the 9th international conference on Autonomic computing*. ACM. 2012, 191–200.

[18] C. Esteban, S. L. Hyland and G. Rätsch. Real-valued (medical) time series generation with recurrent conditional gans. *arXiv preprint arXiv:1706.02633* (2017).

[19] G. Forestier, F. Petitjean, H. A. Dau, G. I. Webb and E. Keogh. Generating synthetic time series to augment sparse datasets. *2017 IEEE international conference on data mining (ICDM)*. IEEE. 2017, 865–870.

[20] Y. Freund and R. E. Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences* 55.1 (Aug. 1997), 119–139.

[21] J. H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics* 29 (2000), 1189–1232.

[22] D. Gannon, R. Barga and N. Sundaresan. Cloud-native applications. *IEEE Cloud Computing* 4.5 (2017), 16–21.

[23] P. Geurts, D. Ernst and L. Wehenkel. Extremely randomized trees. *Machine Learning* 63.1 (Apr. 2006), 3–42.

[24] Q. Guan, Z. Zhang and S. Fu. Ensemble of bayesian predictors for autonomic failure management in cloud computing. *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2011, 1–6.

[25] 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* 94 (2016), 491–496.

[26] Z. Guo, G. Jing, H. Chen and K. Yoshihira. Tracking Probabilistic Correlation of Monitoring Data for Fault Detection in Complex Systems. *Int. Conf. on Dependable Systems and Networks*. 2006, 259–268.

[27] M. Hahmann, C. Hartmann, L. Kegel and W. Lehner. Large-Scale Time Series Analytics. *Datenbank-Spektrum: Vol. 19, No. 1*. Berlin Heidelberg: Springer Nature, 2019, 17–29.

[28] K. G. Hartmann, R. T. Schirrmeister and T. Ball. EEG-GAN: Generative adversarial networks for electroencephalograhic (EEG) brain signals. *arXiv preprint arXiv:1806.01875* (2018).

[29] K. Hightower, B. Burns and J. Beda. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. " O'Reilly Media, Inc.", 2017.

[30] J. Hochenbaum, O. Vallis and A. Kejariwal. Automatic Anomaly Detection in the Cloud Via Statistical Learning. eprint. *arXiv* 1704 (2017).

[31] O. Ibidunmoye, F. Hernández-Rodriguez and E. Elmroth. Performance Anomaly Detection and Bottleneck Identification. *ACM Comput. Surv.* 48.1 (July 2015), 4:1–4:35.

[32] O. Ibidunmoye, F. Hernández-Rodriguez and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)* 48.1 (2015), 4.

[33] O. Ibidunmoye, T. Metsch and E. Elmroth. Real-time detection of performance anomalies for cloud services. *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*. IEEE. 2016, 1–2.

[34] O. Ibidunmoye, A.-R. Rezaie and E. Elmroth. Adaptive anomaly detection in performance metric streams. *IEEE Transactions on Network and Service Management* 15.1 (2017), 217–231.

[35] T. Idé and H. Kashima. Eigenspace-based Anomaly Detection in Computer Systems. *Int. Conf.on Knowledge Discovery and Data Mining*. 2004.

[36] N. Iftikhar, X. Liu, S. Danalachi, F. E. Nordbjerg and J. H. Vollesen. A scalable smart meter data generator using spark. *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2017, 21–36.

[37] S. Jin, Z. Zhang, K. Chakrabarty and X. Gu. Changepoint-based anomaly detection in a core router system. *International Test Conference (ITC)*. Oct. 2017, 1–10.

[38] F. Junqueira and B. Reed. *ZooKeeper: distributed process coordination*. " O'Reilly Media, Inc.", 2013.

[39] A. Kafka. A high-throughput distributed messaging system. *URL: kafka. apache. org as of* 5.1 (2014).

[40] Y. Kang, R. J. Hyndman and K. Smith-Miles. Visualising forecasting algorithm performance using time series instance spaces. *International Journal of Forecasting* 33.2 (2017), 345–358.

[41] Y. Kang, R. J. Hyndman, F. Li et al. *Efficient generation of time series with diverse and controllable characteristics*. Tech. rep. Monash University, Department of Econometrics and Business Statistics, 2018.

[42] L. Kegel, M. Hahmann and W. Lehner. Feature-driven Time Series Generation. *Grundlagen von Datenbanken*. 2017, 54–59.

[43] L. Kegel, M. Hahmann and W. Lehner. Generating what-if scenarios for time series data. *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. ACM. 2017, 3.

[44] L. Kegel, M. Hahmann and W. Lehner. Template-based Time Series Generation with Loom. Citeseer.

[45] J. Kreps, N. Narkhede, J. Rao et al. Kafka: A distributed messaging system for log processing. *Proceedings of the NetDB*. 2011, 1–7.

[46] A. Lakhina, M. Crovella and C. Diot. Diagnosing network-wide traffic anomalies. *ACM SIGCOMM computer communication review*. Vol. 34. 4. ACM. 2004, 219–230.

[47] N. Laptev. *AnoGen: Deep Anomaly Generator*. Tech. rep. Technical Report. Facebook. https://research. fb. com/wp-content/uploads . . ., 2018.

[48] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang et al. Photo-realistic single image super-resolution using a generative adversarial network. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, 4681–4690.

[49] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems* 24.6 (2013), 1245–1255.

[50] C. Monni, M. Pezzè and P. Gaetano. An RBM Anomaly Detector for the Cloud. *International Conference on Software Testing*. 2019.

[51] N. Narkhede, G. Shapira and T. Palino. *Kafka: the definitive guide: real-time data and stream processing at scale*. " O'Reilly Media, Inc.", 2017.

[52] T. Parr, K. Turgutlu, C. Csiszar and J. Howard. Beware Default Random Forest Importances. *March* 26 (2018), 2018.

[53] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[54] T. Pesch, S. Schröders, H. Allelein and J. Hake. A new Markov-chain-related statistical approach for modelling synthetic wind power time series. *New journal of physics* 17.5 (2015), 055001.

[55] D. M. W. Powers. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies* 2.1 (2011), 37–63.

[56] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele and H. Lee. Generative adversarial text to image synthesis. *arXiv preprint arXiv:1605.05396* (2016).

[57] S. Roy, A. C. König, I. Dvorkin and M. Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, 1167–1178.

[58] D. E. Rumelhart, G. E. Hinton, R. J. Williams et al. Learning representations by back-propagating errors. *Cognitive modeling* 5.3 (1988), 1.

[59] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Softw. Engg.* 14.2 (2009), 131–164.

[60] N. Saarimäki, F. Lomio, V. Lenarduzzi and D. Taibi. Does migrate a monolithic system to microservices decreases the technical debt?: *arXiv preprint arXiv:1902.06282* (2019).

[61] C. Sauvanaud, M. Kaâniche, K. Kanoun, K. Lazri and G. D. S. Silvestre. Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned. *Journal of Systems and Software* 139 (2018), 84–106.

[62] J. Schaffner and T. Januschowski. Realistic tenant traces for enterprise DBaaS. *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2013, 29–35.

[63] R. E. Schapire. The Strength of Weak Learnability. *Machine Learning* 5.2 (1990), 197–227.

[64] G. Smrithy and R. Balakrishnan. A statistical technique for online anomaly detection for big data streams in cloud collaborative environment. *2016 IEEE International Conference on Computer and Information Technology (CIT)*. IEEE. 2016, 108–111.

[65] M. Solaimani, M. Iftekhar, L. Khan and B. Thuraisingham. Statistical technique for online anomaly detection using spark over heterogeneous data from multi-source

vmware performance data. *2014 IEEE International Conference on Big Data (Big Data)*. IEEE. 2014, 1086–1094.

[66] D. Taibi and V. Lenarduzzi. On the definition of microservice bad smells. *IEEE software* 35.3 (2018), 56–62.

[67] D. Taibi, V. Lenarduzzi and C. Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing* 4.5 (2017), 22–32.

[68] D. Taibi, V. Lenarduzzi and C. Pahl. Architectural Patterns for Microservices: A Systematic Mapping Study. *CLOSER*. 2018, 221–232.

[69] D. Taibi, V. Lenarduzzi and C. Pahl. Continuous Architecting with Microservices and DevOps: A Systematic Mapping Study. *International Conference on Cloud Computing and Services Science*. Springer. 2018, 126–151.

[70] D. Taibi, V. Lenarduzzi and C. Pahl. Microservices Anti-Patterns: A Taxonomy. *arXiv preprint arXiv:1908.04101* (2019).

[71] D. Taibi and K. Systä. A Decomposition and Metric-Based Evaluation Framework for Microservices. *arXiv preprint arXiv:1908.08513* (2019).

[72] D. Taibi and K. Systä. From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining. *8th International Conference on Cloud Computing and Services Science, CLOSER*. 2019.

[73] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. *2012 IEEE 32nd International Conference on Distributed Computing Systems*. IEEE. 2012, 285–294.

[74] K. Thein. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research* 3.47 (2014), 9478–9483.

[75] M. Thill, W. Konen and T. Bäck. Online anomaly detection on the webscope S5 dataset: A comparative study. *2017 Evolving and Adaptive Intelligent Systems (EAIS)*. IEEE. 2017, 1–8.

[76] H. D. Vinod, J. López-de-Lacalle et al. Maximum entropy bootstrap for time series: the meboot R package. *Journal of Statistical Software* 29.5 (2009), 1–19.

[77] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield and K. Schwan. Statistical techniques for online anomaly detection in data centers. *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*. IEEE. 2011, 385–392.

[78] H. Yoon, K. Yang and C. Shahabi. Feature subset selection and feature ranking for multivariate time series. *IEEE transactions on knowledge and data engineering* 17.9 (2005), 1186–1198.