FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title: Anomaly detection on the CERN data centre monitoring data
Student: Bc. Antonín Dvořák
Supervisor: Mgr. Alexander Kovalenko, Ph.D.
Study program: Informatics
Branch / specialization: Knowledge Engineering
Department: Department of Applied Mathematics
Validity: until the end of summer semester 2021/2022

Instructions

Using time-series classification and forecasting algorithms extend the current anomaly detection alarming system used in the Data Centre at CERN.

The objectives of this work are:

- Analyze the latest state-of-the-art approaches for anomaly detection and forecasting using deep neural network solutions.
- Train, test, and compare the most promising models on the given data.
- Extend the current threshold-based alarming system to efficiently and promptly identify real issues.
- Explore the possibility of anomaly forecasting using deep neural network solutions.
- Implement an Expert feedback loop.
- Create a labeled dataset of Data Centre metrics to be used by the Machine Learning community to benchmark future novel Anomaly Detection algorithms.

The work will be co-supervised and consulted by RNDr. Dagmar Adamová, CSc. (Nuclear Physics Institute of the Czech Academy of Sciences (NPI)), workflow will be conducted in cooperation with CERN and NPI.

Electronically approved by Ing. Karel Klouda, Ph.D. on 21 January 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Anomaly detection on the CERN data centre monitoring data

Bc. Antonín Dvořák

Department of Applied Mathematics
Supervisor: Mgr. Alexander Kovalenko, Ph.D.
Co-supervisor: RNDr. Dagmar Adamová, CSc.[1]
Co-supervisor: Domenico Giordano, Ph.D.[2]

May 4, 2022

[1]Nuclear Physics Institute of the Czech Academy of Sciences
[2]CERN, IT-CM-RPS

Acknowledgements

I gratefully acknowledge the effort of everyone who had supported me on this journey, especially my family and girlfriend.

In the scope of the thesis, I would like to thank Alex for his valuable insights. Thank you, Dagmar, and the whole Nuclear Physics Institute of the CAS, for this opportunity and support. Thanks to the CERN Cloud Infrastructure section (IT-CM-RPS) for welcoming me into the team and mostly to Domenico for guiding me through this project. Thanks to Stiven, who has been working with me on the project while doing his thesis. Last but not least, thanks to Matteo for building the solid foundation, on which this project is built on.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 4, 2022 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Antonín Dvořák. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Dvořák, Antonín. *Anomaly detection on the CERN data centre monitoring data.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Jednou z mnoha úloh CERN cloud manažerů je zajistit požadovaný výpočetní výkon všem uživatelům dané vědecké komunity. Toho je dosaženo pečlivě nastaveným statickým alarming systémem nad výkonostními metrikami infrastruktury.

Pro dosažení maximální efektivity cloudové infrastruktury a ulehčení práce cloud operátorům jsme vytvořili plně automatizovaný systém pro detekci anomálií, který využívá metody nesupervizovaného učení nad časovými řadami. Konkrétně používá kombinaci tradičních metod strojového učení (Isolation forest) a metod hlubokého učení (Gated recurrent unit/Long short-term memory autoencodery).

Tato práce zahrnuje popis monitorovací infrastruktury CERNU, formulaci problému, design systému pro detekci anomálií, použité modely, tvorbu datasetu a porovnání výsledků implementovaných modelů vůči aktuálnímu alarming systému.

Klíčová slova detekce anomálií, nesupervizované strojové učení, časové řady, cloud

Abstract

One of the many tasks of CERN cloud service operators is to make sure that the desired computational power is delivered to all users of the scientific community. This task is accomplished by carefully setting threshold-based alarming on top of the infrastructure performance time series metrics.

In order to maximize the efficiency of the cloud infrastructure and to reduce the monitoring effort for service operators, we have developed a fully automated Anomaly Detection System that leverages unsupervised machine learning methods for time series metrics. Moreover, adopting ensemble methods, we combine traditional (Isolation forest) and deep learning (Gated recurrent unit/Long short-term memory Autoencoders) approaches.

This work presents a description of the CERN monitoring infrastructure, problem formulation, design of the Anomaly Detection Pipeline, description of used models, creation of the dataset and performance of the implemented models compared to the performance of the Current Alarming System.

Keywords anomaly detection, unsupervised machine learning, time series, cloud

Contents

Introduction **1**
CERN Data Centre . . . 2
Thesis Outline . . . 3

1 Monitoring Infrastructure **5**
1.1 MONIT Architecture . . . 5
1.1.1 Collection . . . 6
1.1.2 Ingestion . . . 6
1.1.3 Transport/Processing . . . 6
1.1.4 Storage . . . 7
1.1.5 Presentation . . . 7
1.2 MONIT Technologies . . . 7
1.2.1 Collectd . . . 7
1.2.2 Spark . . . 8
1.2.3 HDFS . . . 8
1.2.4 ELK Stack . . . 8
1.2.5 Grafana . . . 8
1.2.6 SWAN . . . 9

2 Problem Formulation **11**
2.1 Definitions . . . 11
2.1.1 Time Series . . . 11
2.1.2 Metric . . . 11
2.1.3 Hostgroup . . . 12
2.1.4 Anomaly . . . 12
2.1.5 Anomaly Detection . . . 12
2.2 Anomaly Detection on the monitoring data . . . 13
2.2.1 Anomaly in data centre context . . . 14

3 Anomaly Detection Pipeline **15**

3.1 Input preparation 15
3.1.1 Filtering 17
3.1.2 Aggregation 17
3.1.3 Normalization 17
3.1.4 Windowing 18
3.2 Training 18
3.3 Inference 19
3.4 Publish 19
3.5 Presentation 20
3.6 Production deployment 20
3.6.1 Airflow 20
3.6.2 GitLab CI/CD 22

4 Anomaly Detection Models 23
4.1 Traditional Machine Learning 23
4.1.1 Isolation Forest 24
4.2 Deep Learning 25
4.2.1 LSTM Autoencoder 26
4.2.2 GRU Autoencoder 29
4.3 Ensemble 30

5 Labeled Dataset 33

6 Experiments and Results 35
6.1 Used Resources 36
6.2 Input metrics 36
6.3 Figure of Merit 37
6.3.1 AUC-ROC 37
6.3.2 Training and Inference time 38
6.4 Performance of the Individual Models 39
6.4.1 Isolation Forest 39
6.4.2 Autoencoders 40
6.4.3 Summary 44
6.5 Comparison with Current Alarming System 45
6.5.1 Current Alarming System 45
6.5.2 Evaluation and Comparison 46

7 Future Work 49

Conclusion 51

Bibliography 53

Acronyms 57

A Contents of enclosed CD **59**

List of Figures

1.1 MONIT architecture [1] 6

2.1 Long-term load metric for multiple HVs 12

3.1 Anomaly Detection System architecture 16
3.2 Example of Grafana dashboard with anomaly scores 20
3.3 Airflow DAG for the pipeline 21
3.4 Airflow scheduling over multiple days 22
3.5 GitLab CI/CD pipeline 22

4.1 Partitioning of the 2D feature space and Path Length in Isolation Tree for anomalous and normal sample; individual cuts required to isolate given sample are numbered [2] 25
4.2 LSTM Cell [3] 28
4.3 GRU Cell [4] 30

5.1 Labeled Dataset based on a Shared Hostgroup: blue = normal, orange = anomaly 34

6.1 Confusion Matrix 37
6.2 ROC and AUC of poorly performing model (left) and well-performing model (right) 38
6.3 Training/validation loss and distribution of the scores of poorly performing GRU model (5 Units, 0 Dropout) with AUC-ROC of 0.617 41
6.4 Training/validation loss and distribution of the scores of well-performing GRU model (1 Unit, 0.25 Dropout) with AUC-ROC of 0.967 . . . 42
6.5 CPU Load alerts in the Current Alarming System: horizontal red line = threshold, vertical red arrow = alert triggered, vertical green arrow = previous alert no longer triggered 46

6.6 Grafana Dashboard containing anomaly scores from ensemble method ENS_3 and individual models for the time period of 7 days 48

List of Tables

6.1 Monitoring metrics used by the proposed solution and used by the current alerting system. 36
6.2 AUC-ROC, training and inference time of IFOR with different numbers of iTrees 39
6.3 AUC-ROC, training and inference time of IFOR with different lengths of the training period 40
6.4 AUC-ROC, training, and inference time of LSTM-AE with different parameters 40
6.5 AUC-ROC, training, and inference time of GRU-AE with different parameters 41
6.6 AUC-ROC, training and inference time of LSTM-AE with different lengths of the training period 43
6.7 AUC-ROC, training and inference time of GRU-AE with different lengths of the training period 43
6.8 AUC-ROC of LSTM-AE and GRU-AE with a single Dropout layer 43
6.9 AUC-ROC of LSTM-AE proposed in previous work [5] 44
6.10 Stability of model performance based on the average and standard deviation of the AUC-ROC for one week-long (AUC-ROC week) and one month-long (AUC-ROC Month) training periods 44
6.11 True positive rate (TPR) measured at different values of the False Positive Rate (FPR) for the predictions of the Current Alarming System, the proposed individual models, and the proposed ensemble methods ENS_e for $e = 1, 2, 3$ 47

Introduction

"The Big Bang should have created equal amounts of matter and antimatter in the early universe. But today, everything we see from the smallest life forms on Earth to the largest stellar objects is made almost entirely of matter. Comparatively, there is not much antimatter to be found. Something must have happened to tip the balance. One of the greatest challenges in physics is to figure out what happened to the antimatter, or why we see an asymmetry between matter and antimatter?" [6]

This is the type of questions that CERN, the European Organisation for Nuclear Research, is aiming to answer. CERN is the largest particle physics laboratory in the world, located on the border of France and Switzerland. CERN has built and is operating Large Hadron Collider (LHC) which is the world's largest and most powerful particle accelerator.

Processing the data produced by the LHC experiments requires a significant amount of computing power and storage. During previous years the experiments produced up to 90 petabytes of data per year. Towards the high multiplicity LHC upgrade, this amount will significantly increase. To process and store the LHC data, CERN has built a worldwide infrastructure involving almost 170 computing sites in 42 countries spread over 5 continents interconnected by a high capacity network, the Worldwide LHC Computing Grid (WLCG) [7].

CERN Data Centre

Let's begin with an introduction to OpenStack, Hypervisor, and Virtual Machine:

OpenStack: a cloud operating system that controls large pools of computing, storage, and networking resources throughout a data centre, all managed and provisioned through Application Programming Interface (API)s with common authentication mechanisms [8].

Hypervisor: software that creates and runs Virtual Machine (VM). It isolates the hypervisor's operating system and resources from the VM and manages the lifecycle of the VMs [9].

Virtual machine: a virtual environment that functions as a virtual computer system with its own Central Processing Unit (CPU), Random-Access Memory (RAM), network interfaces, and storage, created on physical hardware [10].

The CERN data centre is in the middle of its scientific infrastructure and is the central site of the WLCG. Over 90% of the computing resources at CERN are provided by OpenStack private cloud, which has been deployed by the CERN IT department and has been running in the production environment since 2013 [11].

This cloud is composed of 7895 physical servers, which contain 423 000 CPU cores and 1,73 petabytes of RAM.

1695 of the servers are used as a Hypervisor (HV), which contain 48 100 CPU cores and 296 terabytes of RAM. They are hosting 14 435 VMs, which are used for computations in various physics experiments and in internal IT services. The rest of the servers are used for the provisioning of bare metal machines instead of VMS.

Compared to previous years the number of HVs has drastically decreased – in January 2021 there were around 8000 of them. The introduction of OpenStack Ironic project at CERN allowed migration of VMs to physical servers, which can be operated through the same OpenStack API as VMs, with the benefit of increased performance (approximately 5% gain from the removal of the virtualization tax [12]). The CERN Batch service used this possibility and migrated most of its VMs to bare metals in multiple chunks over several months.

During the cooperation with CERN, I was part of the *IT-CM-RPS* (IT department, Compute & Monitoring, section of Resource Provisioning Services), which is responsible for the maintenance, operations, and provisioning of the Cloud Infrastructure.

Thesis Outline

The thesis is structured as follows:

- Chapter 1 describes the CERN Monitoring Infrastructure,
- Chapter 2 defines important terms and presents the problem formulation,
- Chapter 3 describes the structure of the Anomaly Detection Pipeline,
- Chapter 4 introduces models used for the Anomaly Detection,
- Chapter 5 presents a labeled dataset,
- Chapter 6 contains experiments for the proposed solution and its comparison with respect to the Current Alarming System,
- Chapter 7 summarizes topics that can be explored in the future.

Chapter 1

Monitoring Infrastructure

In this chapter, we will introduce the architecture of the Monitoring Infrastructure and a more in-depth description of several used technologies relevant to the scope of this work.

Monitoring is an essential component (especially) in large-scale infrastructure. It is providing the users with information about the state of their service(s) at a given point in time. It can provide information for both software and hardware services. Monitoring data consist of:

- metrics – state information,
- logs – event records,
- alarms – notifications about the abnormal state.

1.1 MONIT Architecture

MONIT is a Unified Monitoring that was introduced in 2016 when it was decided to merge the WLCG monitoring services, resources, and technologies with the internal CERN IT data centre monitoring services [13].

The monitoring infrastructure is very robust. It has to work with data provided by all servers, both virtual and physical. This comes to around 20 000 providers, which are sending different kinds of metrics and logs. This equals to more than 3 terabytes of compressed data every day, with an input rate of 100 kHz [14].

The architecture of MONIT can be seen in Figure 1.1. All parts will be introduced in the following sections. The architecture description is based on multiple sources [1, 15, 16].

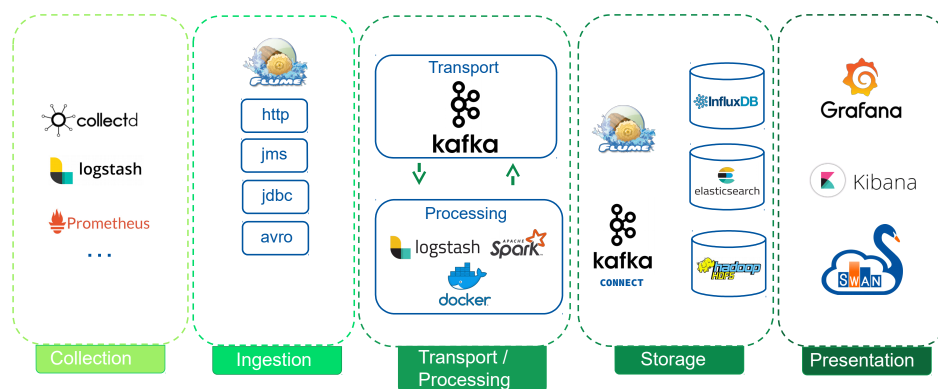


Figure 1.1: MONIT architecture [1]

1.1.1 Collection

The collection layer is composed of multiple daemon agents collecting relevant monitoring data. These agents are executed locally on the monitored hosts. They collect Operating System (OS) and application metrics, logs and alarms. Their job is to collect the data and send results to the next layer.

Various open-source technologies are used for the data collection, for example Collectd, Fluentd/Logstash (component of ELK Stack), Prometheus and the selection depends on specific use case.

1.1.2 Ingestion

The ingestion layer provides various inlet points (gateways) for different data collectors. Most of the data is sent to Apache Flume which supports 14 different gateways (collectd, Prometheus, JavaScript Object Notation (JSON), etc.) and is composed of more than 200 instances. The goal of the ingestion layer is to prepare data with a predefined scheme.

1.1.3 Transport/Processing

The transport/processing layer is the divider between producers and consumers.

The transport part enables streaming processing and provides resilience and reliability. It is provided by Apache Kafka, which is a distributed stream-

ing platform. It runs on 20 instances with triplicated data and offers a buffer with a retention period of 72 hours.

The processing part offers processing over big datasets (correlation, aggregation, transformation, data enrichment, compression, etc.). Data can be streamed directly from Kafka or it can be read from storage. Processing is mostly done using Spark.

1.1.4 Storage

The storage layer provides multiple backends, which support different retention periods and types of data.

Time series can be stored in InfluxDB, which offers 3 retention periods – 1 week for original data, 1 month with aggregation of 5 minutes, and 5 years with aggregation of 1 hour. Elasticsearch (the main component of ELK Stack) is used to store logs and other JSON documents for a short-term period. HDFS is used as long-term storage with no aggregation.

1.1.5 Presentation

The presentation layer defines, how users can observe the data and is composed of multiple web applications. It offers dashboards, notifications, and search tools.

The most common tool is Grafana for visualizations of time series obtained from both InfluxDB and Elasticsearch. Kibana (a component of ELK Stack) is an alternative to Grafana for observations of Elasticsearch documents. SWAN offers an interactive data analysis.

1.2 MONIT Technologies

In this section, we will describe some of the used technologies in the MONIT, that are relevant to the scope of this work and are used in the Anomaly Detection Pipeline.

1.2.1 Collectd

Collectd is a lightweight Unix daemon introduced in 2005 that collects, stores, and transfers performance data [17]. It is a modular solution based on plugins. The output of a single plugin can consist of multiple parameters. There is a variety of out-of-the-box plugins, mostly for OS metrics, but also for some application metrics. Custom plugins can be made for specific use cases as long as they follow the required output structure.

1.2.2 Spark

Apache Spark is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters [18].

Spark application consists of two main components:

- driver – converts the user's code into multiple tasks that can be distributed across worker nodes
- executor – executes the assigned tasks

1.2.3 HDFS

Hadoop Distributed File System (HDFS) is a distributed file system, which is highly fault-tolerant and is designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications that operate with large datasets [19].

1.2.4 ELK Stack

ELK Stack is a set of tools for centralized logging, optimized for fast search. ELK is an acronym for three open source projects: Elasticsearch, Logstash, and Kibana.

Logstash is a tool to collect, process, and forward events/log messages. When the input plugin collects data, it can be processed by multiple filters, which modify and annotate the data. Then Logstash transports data to output plugins, which can forward them to various external programs, for example, Elasticsearch.

Elasticsearch is a distributed, scalable, and highly available search and analytics engine, excelling at full-text search in near-real-time (meaning document is indexed and searchable within seconds) built on top of Apache Lucene. Elasticsearch is considered a NoSQL database.

Kibana is a browser-based data visualization dashboard for Elasticsearch. It is used for searching, viewing, visualizing, and analyzing the data from Elasticsearch [20].

1.2.5 Grafana

Grafana is web-based visualization and analytics software introduced in 2014. It is capable to query multiple data sources, creating dashboards and alerts based on some evaluation criteria.

Supported data sources are for example InfluxDB, Elasticsearch, and Prometheus.

1.2.6 SWAN

Service for Web based Analysis (SWAN) is a CERN web-based platform to perform interactive data analysis with a Jupyter notebook interface and shell access without the need to install any software. Data can be stored in the personal or shared part of the cloud storage, which makes sharing easy. SWAN is capable of offloading big data computations to Spark [21].

Chapter **2**

Problem Formulation

In this chapter, we will define some of the essential terms and formulate the problem of Anomaly Detection on monitoring data time series.

2.1 Definitions

Let's start with the definition of generic terms, that are further important for the specific problem formulation.

2.1.1 Time Series

A time series is an ordered sequence of numerical data points measured over successive times. It is defined as $\{X_t\}, t \in T$. Set $T = \{t_0, t_1, \ldots\}$ represents time of measurement and variable X_t is a time-dependent random variable.

A time series having a single time-dependent variable is called *univariate*. Time series containing multiple time-dependent variables is called *multivariate*.

Time series can be *continuous* or *discrete*. Continuous time series represent a data point for every point in time. On the other hand, discrete time series consist of discrete points in time, usually, the consecutive observations are spaced by fixed time intervals.

2.1.2 Metric

Metric is a *discrete univariate* time series, which represents the behavior of a single parameter collected by collectd plugin.

Figure 2.1 displays one week of *long-term load* metric reported by collectd Load plugin for HVs in a single Hostgroup (HG).

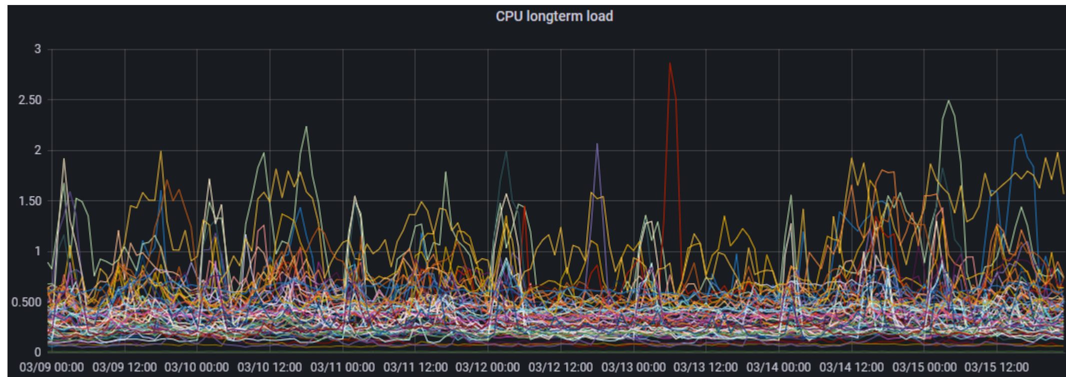


Figure 2.1: Long-term load metric for multiple HVs

2.1.3 Hostgroup

A Hostgroup is a set of HVs having the same configuration and hardware resources. In this work, we observe Hostgroups called *shared* – a set of HVs shared by service servers. These HGs are considered as most complex and most difficult to monitor with threshold-based alerting due to the high variability.

2.1.4 Anomaly

In spoken language, an anomaly is something different, abnormal, peculiar, or not easily classified. Something which deviates from the common rule [22].

In terms of data science, anomalies are rare patterns in data that significantly differ from normal behavior. They might be present in the data for many reasons, for example, due to malicious activity (credit card fraud, cyber-intrusion, terrorist activity, breakdown of a system, etc.) [23]. To be able to declare what normal behavior is we need a dataset, thus anomalies are dataset-dependant deviations.

In general, we can't say anomalies are exclusively good or bad. They are just deviations from the expected behavior at a given point in time.

The consequences of deviant behavior are the driving force of Anomaly Detection (AD).

2.1.5 Anomaly Detection

Anomaly Detection is a data processing technique to detect anomalies in the dataset. In terms of machine learning, it can be *supervised*, *semi-supervised* or *unsupervised*. Most of the time it is applied to unlabeled datasets, which makes it unsupervised AD.

The goal of unsupervised AD is to detect deviating patterns without any prior knowledge about what is or isn't anomalous. The assumption is that

the percentage of anomalies in the dataset is small (usually less than 1%) and they are significantly different from what can be considered normal behavior. In most cases, this means modeling normal data distribution and defining measurements in this space to be able to classify samples as normal or anomalous [24].

2.2 Anomaly Detection on the monitoring data

We define a data centre as a set of computer clusters. Cluster C is composed of N HVs having the same hardware equipment and software configuration (for example HVs from the same HG). Each specific HV in the cluster is denoted by h_i with $i = 1, \ldots, N$ and generates M metrics, such as the usage of CPU, memory, disk, network, etc. We denote with

$$m_i^j = \{m_i^j(t), t = T_1, T_2, \ldots\} \tag{2.1}$$

the jth metric for the HV h_i measured at the timestamps t, where t is arbitrarily numbered from 1 to infinite.

All the metrics $\{m_i^j, i = 1, \ldots, N, j = 1, \ldots, M\}$ might be measured at different times $T_1, T_2, \ldots$, possibly with different sampling frequencies (e.g., *Disk IO Time* is being acquired at a frequency in the order of seconds, while *CPU Load* in the order of minutes). However, we assume that all these are synchronized, namely that they refer to an identical reference clock.

We assume that in normal operating conditions, all metrics are being generated by a normal (non-anomalous) process $\mathcal{P}_N$ which is unknown. If at time τ an anomaly occurs in the data centre, the metrics are generated by an anomalous process $\mathcal{P}_A$, which is also unknown and different from $\mathcal{P}_N$. The anomalous behavior might affect only a few HVs or a few metrics, thus we model our metrics as follows:

$$m_i^j(t) \sim \begin{cases} \mathcal{P}_N & \text{if } t < \tau \ \ \forall i,\ j, \\ \mathcal{P}_A & \text{if } t \geq \tau \text{ for at least some } i,\ j, \end{cases} \tag{2.2}$$

where we assume that the metrics m_i^j that are not affected remain generated by $\mathcal{P}_N$ even if $t \geq \tau$. Even though (2.2) describes a change-detection problem, we refer to Anomaly Detection in this work since the process $\mathcal{P}_A$ is typically triggered by a HV following an anomalous behavior.

Our goal is to design a model that steadily monitors all metrics $\{m_i^j, i = 1, \ldots, N, j = 1, \ldots, M\}$ and that promptly reports any anomaly. For this purpose, we assume that a training set TR, containing a portion of the time series from all the N HVs of the cluster C, is given to configure our model. TR is not guaranteed to be free from anomalies. However, from empiric evidence, we assume that the vast majority of TR is generated by $\mathcal{P}_N$, and anomalies – if any – are very rare. TR is unlabeled, therefore we operate in an unsupervised Anomaly Detection scenario.

2.2.1 Anomaly in data centre context

One of the biggest issues in a virtualized environment is a situation when the state of HV negatively affects the performance of hosted VMs. This can be caused by hardware failures, misconfigurations, or by the *noisy neighbor problem*.

A noisy neighbor [25] is a VM that over-consumes HV resources and leaves other VMs with resource deficit, which will affect the performance. This can be related to the CPU, RAM, network, storage, etc.

Chapter 3

Anomaly Detection Pipeline

Proof of Concept of the Anomaly Detection Pipeline and Library was developed in the previous work [5]. This work is an extension to it, introducing new features, modifications and further optimizations. Our GitLab repository containing the source code is publicly available [3].

The architecture of the system is displayed in Figure 3.1 and the workflow consists of the following steps:

1. Input preparation – read the collectd data from HDFS, prepare it using Spark and move the prepared data to EOS,
2. Training – train the models on training data,
3. Inference – produce anomaly scores on inference data,
4. Publishing – prepare the results and send them to Elasticsearch,
5. Presentation – visualize results using Grafana dashboards.

At the end of the chapter we describe how the production pipeline is deployed.

It is also important to mention that **HDFS does not contain online data**. All the collectd data for a given day are available on the HDFS the next day, usually around 4 AM. In the current state, we present an **offline AD system**, which produces anomaly scores for the data of the previous day and presents them to service managers in the morning of the next day.

3.1 Input preparation

This section covers the preparation of the input data for the AD system. Algorithm 1 presents the pseudocode of this procedure.

[3] https://gitlab.cern.ch/cloud-infrastructure/data-analytics/

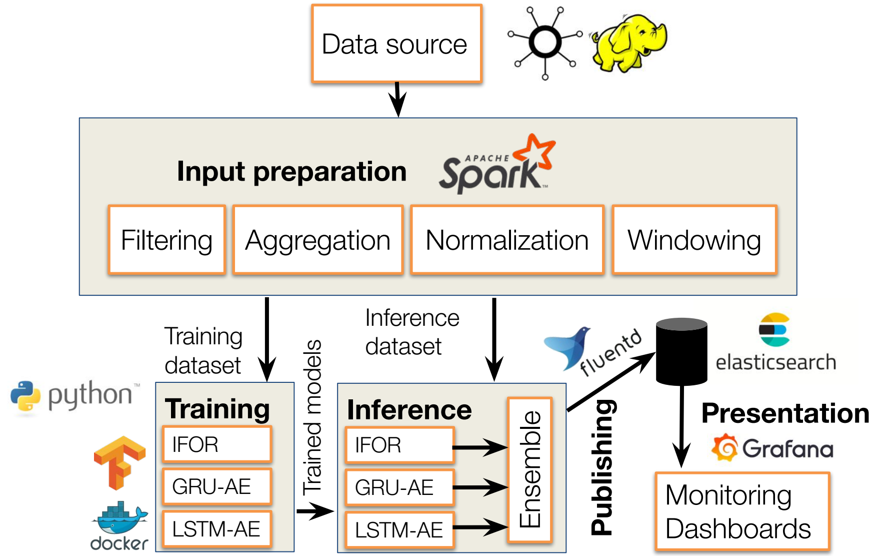


Figure 3.1: Anomaly Detection System architecture

Algorithm 1: Input Preparation Pseudocode

```
   Data: time_start, time_end, aggr_period, window_length, metrics,
         hostgroups
   Result: multivariate_time_windows
1  collectd_plugins = get_plugins(metrics) // get plugin names from
     metrics definition
2  collectd_data = hdfs_read(collectd_plugins, time_start, time_end)
     // load plugins for specified time period
3  filtered_data = filter_data(collectd_data, metrics, hostgroups) // get
     data for specified metrics and HGs
4  aggregated_data = aggregate(filtered_data, aggr_period) // aggregate
     data with given aggregation period
5  norm_metrics = empty list // prepare list for normalized metrics
   // iterate over each metric
6  for each metric m in aggregated_data do
7  |  stdev = get_stdev(m) // calculate standard deviation for metric m
8  |  mean = get_mean(m) // calculate mean for metric m
9  |  norm_metric = normalize(m, stdev, mean) // normalize metric m
10 |  norm_metrics.append(norm_metric) // save normalized metric
11 end
12 multivariate_time_windows = windowing(norm_metrics,
     window_length) // prepare normalized time windows
13 return multivariate_time_windows
```

3.1.1 Filtering

Every HV h produces M (order of hundreds) real-valued metrics m, but only K of them are used for the purpose of AD.

HDFS contains original (unaggregated) Collectd metrics for all hosts and plugins. Filtering process is applied to obtain data in given time period only for selected metrics and N HVs forming the cluster $C = (h_1, h_2, \ldots, h_N)$.

3.1.2 Aggregation

Metrics are collected at different times for each HV h_i and with different frequencies depending on the definition of the plugins. In order to provide consistent measurements, the aggregation has to be applied to the original metrics. The *aggregation period* must not be lower than the lowest collecting frequency of all observed plugins.

Given jth metric $m_i^j = \{m_i^j(t_0), m_i^j(t_1), \ldots\}$ for a HV h_i the aggregated metric is produced by averaging m_i^j values in disjoint consecutive intervals with size of *aggregation period* forming new metric $d_i^j = \{d_i^j(t_0), d_i^j(t_1), \ldots\}$. Formally:

$$d_i^j(t_k) = mean(m_i^j(t) \in m_i^j \mid t_k - aggregation\ period < t \leq t_k). \tag{3.1}$$

3.1.3 Normalization

To make all the metrics equally important in the AD problem, we normalize each of them relatively to the cluster C. We assume that HVs in the same cluster have the same distribution of the same jth metric. For each metric d^j we compute the mean and standard deviation over the cluster:

$$\mu_j(C) = \frac{\sum_{i \in N} \sum_{t_0 < t \leq t_T} d_i^j(t)}{N \cdot T}, \tag{3.2}$$

$$\sigma_j(G) = \sqrt{\frac{\sum_{i \in N} \sum_{t_0 < t \leq t_T} (d_i^j(t) - \mu_j(C))^2}{N \cdot T}}, \tag{3.3}$$

where t_0 and t_T are respectively the start and end timestamp of the training period, T is the total number of time steps in the training period, and N is the number of HVs in cluster C. We compute $\mu_j(C)$ and $\sigma_j(C)$ for all $j \in \{1, \ldots, K\}$. Then for each HV h_i of cluster C and metric j we compute a normalized metric $n_i^j = \{n_i^j(t_0), n_i^j(t_1), \ldots\}$ where:

$$n_i^j(t_k) = \frac{d_i^j(t_k) - \mu_j(C)}{\sigma_j(C)}. \tag{3.4}$$

This normalization is computed over the training period and then it is applied to the training and the inference data.

3.1.4 Windowing

Each of normalized metrics n_i^j for HV h_i is divided into consecutive non-overlapping sub-series of length $L \in \mathbb{R}$. Metric window is formed by merging all K metrics starting from time t_k – where $k \mod L = 0$. The metric window can be represented as a matrix $W_{i,j} \in \mathbb{R}^{K \times L}$:

$$W_{i,j} = \begin{bmatrix} n_i^1(t_k) & n_i^1(t_{k+1}) & \cdots & n_i^1(t_{k+L-1}) \\ n_i^2(t_k) & n_i^2(t_{k+1}) & \cdots & n_i^2(t_{k+L-1}) \\ \vdots & \vdots & \ddots & \vdots \\ n_i^K(t_k) & n_i^K(t_{k+1}) & \cdots & n_i^K(t_{k+L-1}) \end{bmatrix} \tag{3.5}$$

3.2 Training

In the training part, we prepare all models as described in the configuration file. Each of the models is initialized, built, and trained on the training data. Additionally, anomaly score predictions are generated on the training data to obtain the standard deviation and mean of the scores. These values are later used in the inference part to normalize scores, which helps models produce anomaly scores in a similar range centered around zero. In the end, models are saved to the file system and can be later used (repeatedly) in the inference part without re-training.

This procedure is described in Algorithm 2.

Algorithm 2: Training Procedure

```
Data: training_data, model_config
Result: trained_models
// iterate over defined models
1 for each model in model_conf do
2     model = init_model(model.params) // initialize the model with
          specified parameters
3     model.build() // build the model
4     model.train(training_data) // train the model on training data
5     scores = model.predict(training_data) // produce anomaly scores
          over training data
6     model.tr_stdev = get_stdev(scores) // obtain standard deviation of
          anomaly scores
7     model.tr_mean = get_mean(scores) // obtain mean of anomaly
          scores
8     model.save() // save the model
9 end
```

3.3 Inference

In the inference part models are initialized and then the trained models are loaded. Each model goes through all time windows in the inference dataset and produces an anomaly score. The anomaly score is then normalized with the coefficients of training standard deviation and mean. Both original and normalized anomaly scores along with additional information about the window are then saved to the file system.

This procedure is described in Algorithm 3.

Algorithm 3: Inference Procedure

```
  Data: inference_data, model_config
  Result: Anomaly scores for all windows
  // iterate over defined models
1 for each model in model_conf do
2     model = init_model(model.params) // initialize the model with
        specified parameters
3     model.load() // load the trained model
      // iterate over windows in inference data
4     for each window w in inference_data do
5         score = model.predict(w) // produce anomaly score for given
            window
6         normalized_score = normalize(score,tr_stdev, tr_mean)
            // normalize the score with training coefficients
7         save(score, normalized_score, model.info, window.info) // save
            the result
8     end
9 end
```

3.4 Publish

This step covers the processing of anomaly scores obtained from the Inference part.

Saved scores are divided per time window range and per model. The publisher goes through all time windows in a specified time range and iterates over each of the model results to read the scores and decides what time windows to publish based on publishing strategy (TOP K, above the threshold, etc.).

The time windows, which satisfy the publishing criteria, are processed to produce JSON documents containing very detailed information about the time window – timestamps, HV name, HG name, name of the model, used metrics,

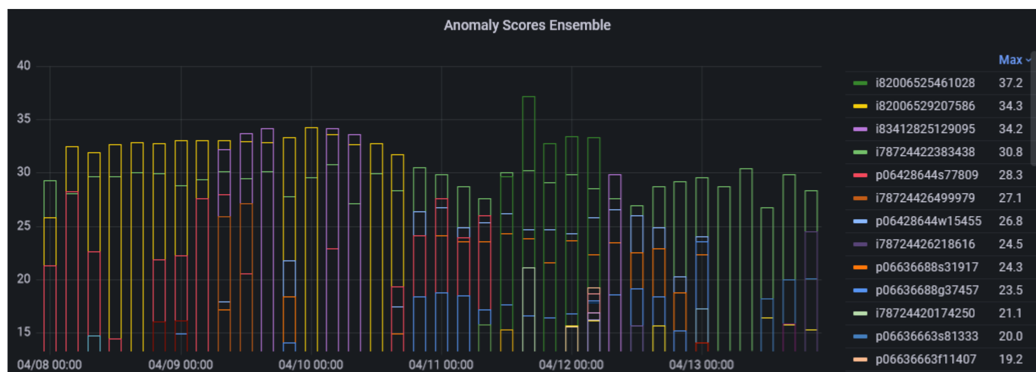


Figure 3.2: Example of Grafana dashboard with anomaly scores

anomaly score and normalized anomaly score, coefficients used to normalize the score, rank of the score, etc. These documents are then sent by Fluentd to Elasticsearch.

3.5 Presentation

Presentation is a step in which results can be visualized and observed.

Presentation consists of multiple sets of Grafana dashboards. Grafana supports Elasticsearch as a data source, which allows us to directly visualize the results prepared by Publish step. Additional aggregation or filtering is supported by Grafana variables and/or queries to obtain only specific entries based on the use case. For each dashboard, there is a possibility to set up alerts to notify cloud operators.

In Figure 3.2 we show an example of a Grafana dashboard displaying sum of normalized anomaly scores above the threshold for three individual models.

3.6 Production deployment

In this section, we will discuss some technologies, which help us to achieve a fully automated AD system and ease the deployment of new features or changes.

3.6.1 Airflow

As an orchestrator, we use Apache Airflow for all the steps of the pipeline. It is an open-source workflow management platform that is configured through

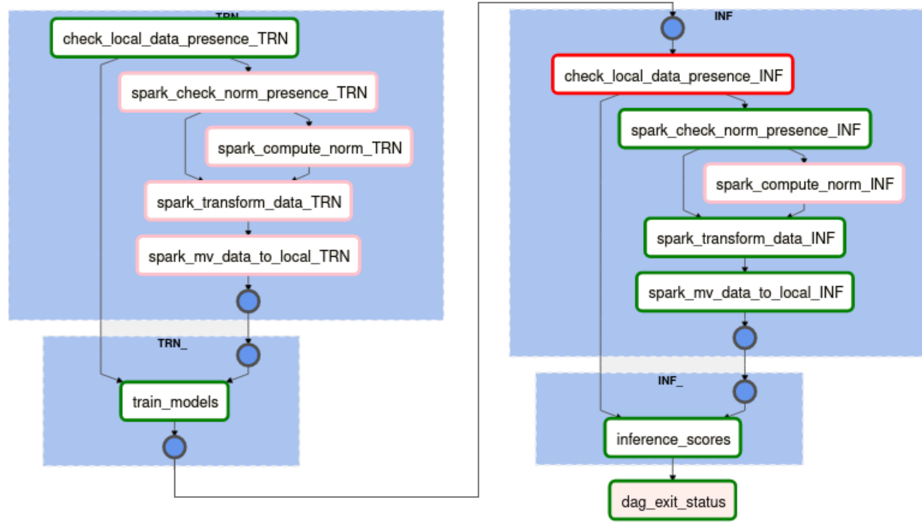


Figure 3.3: Airflow DAG for the pipeline

a Directed Acyclic Graph (DAG). Each node of the DAG represents a step in the pipeline. In Figure 3.3 we show our DAG for the AD pipeline.

Note that green means success, pink skipped, and red failure. Failure does not always mean failure of the pipeline – based on the DAG definition success/failure can be used as a condition to direct the flow of steps.

The DAG is divided into four logical parts:

- **TRN** – check the presence of prepared training data on EOS, if not present then prepare it (Sec. 3.1) and save it to EOS,
- **TRN_** – train the models on training data (Sec. 3.2),
- **INF** – check the presence of prepared inference data on EOS, if not present then prepare it and save it to EOS,
- **INF_** – produce the anomaly scores over inference data (Sec. 3.3) and publish them (Sec. 3.4).

Airflow handles automated scheduling in specified time intervals. In our case, we repeat the pipeline three times a day for the data of the previous day, which is shown in Figure 3.4.

Every step of the pipeline is executed in a Docker container based on images containing all necessary libraries. Docker images are prepared by GitLab Continuous Integration / Continuous Delivery (CI/CD).

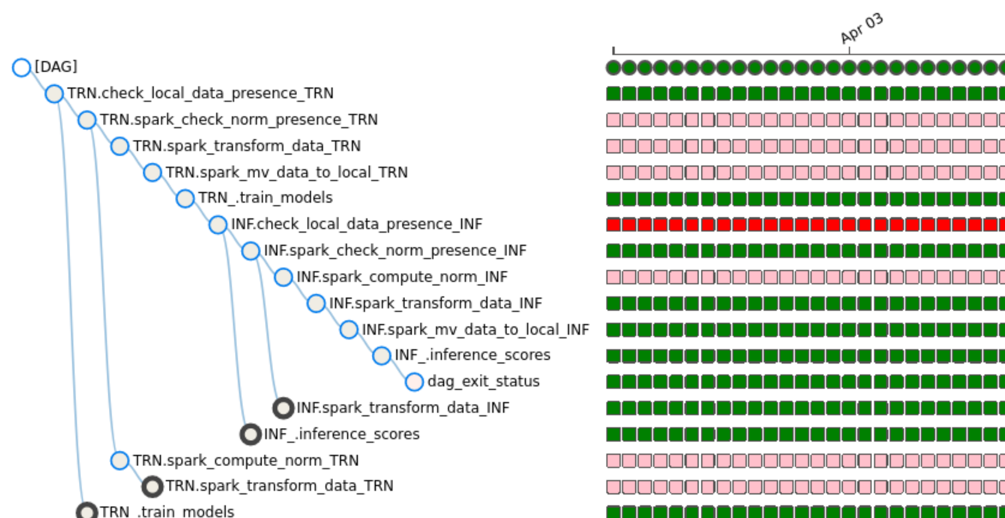


Figure 3.4: Airflow scheduling over multiple days

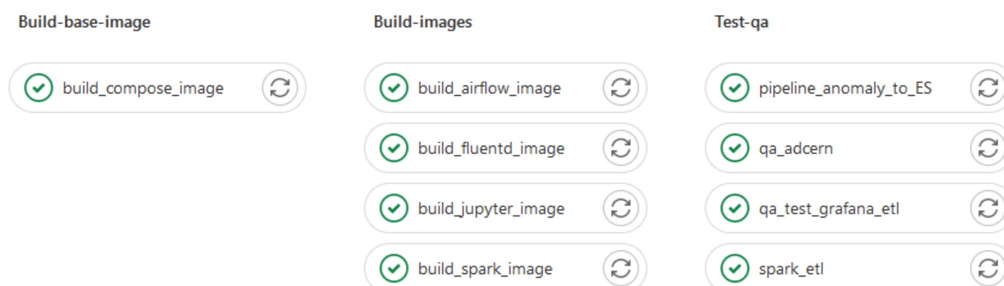


Figure 3.5: GitLab CI/CD pipeline

3.6.2 GitLab CI/CD

We don't use GitLab just to manage the codebase, but we also leverage the usage of the GitLab CI/CD.

When change is applied to specified branches GitLab CI/CD job (Fig. 3.5) is automatically executed. This job is responsible for the preparation of various Docker images and testing of the code changes with a predefined set of test rules to ensure the validity and functionality of the change.

Jobs are also triggered every week even under no changes to ensure no new issues have appeared (for example incompatibilities with upstream libraries).

Chapter **4**

Anomaly Detection Models

We elaborate on the detection of HVs having an anomalous behavior in (2.2) by learning an anomaly score function $\mathcal{A}$ that takes a metric window $W_{i,k}$ (defined in Sec. 3.1.4) as input and returns low scores for $W_{i,j}$ generated by normal process $\mathcal{P}_N$, and higher scores for $W_{i,j}$ containing values generated by anomalous process $\mathcal{P}_A$. The anomaly score function is a mapping of this type:

$$\mathcal{A}(\cdot)\colon \mathbb{R}^{K\times L} \to \mathbb{R}^{+}, \tag{4.1}$$

that analyzes the collective behavior of all N HVs in the cluster C and is trained on the whole training set TR. We detect anomalies by simply comparing $\mathcal{A}(W_{i,k})$ against a threshold Γ as follows:

$$AD(W_{i,k}, \Gamma) = \begin{cases} 1 & \text{if } \mathcal{A}(W_{i,k}) > \Gamma, \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

The threshold Γ is typically set to control the false positive rate. In particular, Γ can be selected as a quantile of the anomaly score from a subset of windows from TR that have not been used for learning $\mathcal{A}$.

Based on the results of previous work [5], the best performing model of Traditional Machine Learning was Isolation Forest (IFOR) and from the Deep Learning Long-Short Term Memory Autoencoder (LSTM-AE). This work extends the study and implements the Deep Learning model Gated Recurrent Unit Autoencoder (GRU-AE).

We have implemented three unsupervised methods (IFOR, LSTM-AE, GRU-AE) to model three different anomaly score functions $\mathcal{A}_\lambda$ ($\lambda = 1, 2, 3$).

4.1 Traditional Machine Learning

Traditional Machine Learning is a subset of Machine Learning algorithms that utilizes a simple structure such as linear regression or decision trees (as opposed to Deep Learning).

Traditional Machine Learning models can be used to solve problems of classification, regression, clustering, dimensionality reduction, and Anomaly Detection.

Examples of Traditional algorithms: Linear Regression, Logistic Regression, Naive Bayes, K-means, Random Forest, Isolation Forest [26].

4.1.1 Isolation Forest

IFOR [27] is an Anomaly Detection algorithm that builds an ensemble of n Isolation Trees. It identifies anomalous samples based on "how early" they can be isolated from the rest of the dataset.

Isolation means the separation of the sample from the rest. It leverages the assumption that anomalies are presented in low numbers and are very different from the normal samples, which allows easier/sooner isolation under random partitioning.

An Isolation Tree (iTree) is a Binary Search Tree (BST), where each node has exactly zero or 2 children nodes, that learns to isolate samples. Each of the n trees is given a dataset $X = \{x_1, \ldots, x_m\}$ and it recursively divides X by randomly selected feature q and split value p until either the tree reaches limit height, X can be no longer divided ($|X| = 1$) or samples in X are equal.

Path Length $h(x)$ of a sample x is defined as the number of edges x has to traverse in an iTree from the root of the tree to the terminal node (node with no child nodes).

Anomaly score can't be easily computed just by the average of $h(x)$ from n trees, because the maximum possible height of iTree grows in the order of m, but the average height grows in the order of $\log m$. Original work [27] exploits the same structure of iTree and BST to define the average path length of iTree as of the BST based on [28] as follows:

$$c(m) = 2H(m-1) - 2(m-1)/m, \tag{4.3}$$

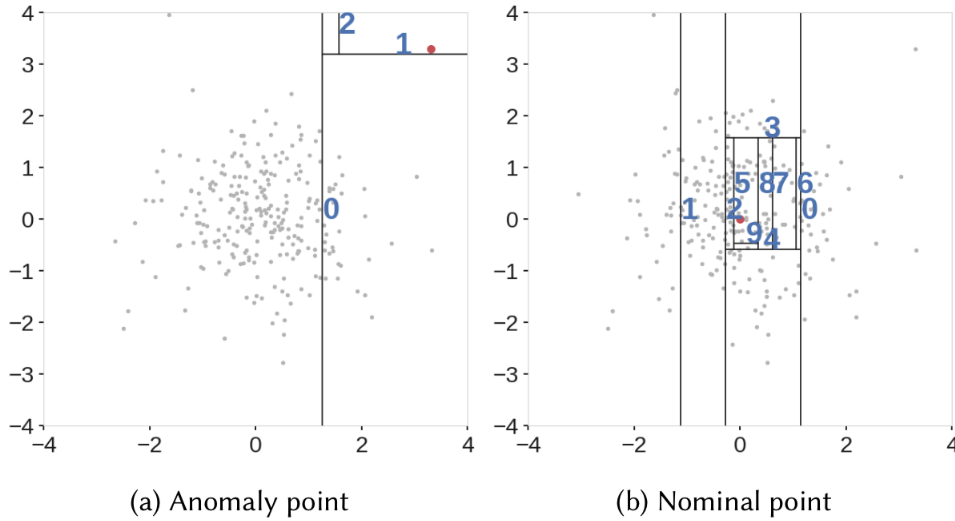
where $H(i)$ is the harmonic number estimated by $\ln i + 0.5772156649$ (Euler's constant). The final anomaly score s of sample x is then defined as:

$$s(x, m) = 2^{-\frac{E(h(x))}{c(m)}}, \tag{4.4}$$

where $E(h(x))$ is the average $h(x)$ over n iTrees. Using the anomaly score s final assumptions are set as:

- if sample returns s very close to 1, then it is definitely anomalous,
- if sample returns s much smaller than 0.5, then it can be safely considered as normal,
- if all samples return s close to 0.5, then X does not contain any anomalous data.

Figure 4.1 shows the difference of Path Length in single iTree for anomalous and normal samples.



(a) Anomaly point (b) Nominal point

Figure 4.1: Partitioning of the 2D feature space and Path Length in Isolation Tree for anomalous and normal sample; individual cuts required to isolate given sample are numbered [2]

Implementation

IFOR is not designed to model the temporal behavior of a multivariate time series. We solve this by concatenating the rows of each window $W_{i,k} \in \mathbb{R}^{K \times L}$ into a vector $\vec{w}_{i,k} \in \mathbb{R}^{V}$ where $V = K \cdot L$, and use this as an input to IFOR.

This work uses the off-the-shelf implementation of PyOD Isolation Forest.

4.2 Deep Learning

Deep Learning is a subset of Machine Learning using a layered structure of artificial neural networks (ANN). This allows a process of learning that's far more capable than traditional machine learning models.

Compared to Traditional Machine Learning it usually requires significantly more computing power and a larger amount of data to be trained [29].

Popular Deep Learning architecture for the purpose of Anomaly Detection is an Autoencoder [30].

Autoencoder

Autoencoder [31] is a type of neural network, that learns to reconstruct the input to its output. It consists of the encoder function ϕ that maps the original input $\mathcal{X}$ to the latent space $\mathcal{F}$, and then reconstructs $\mathcal{X}$ by mapping the decoder function ψ to the latent space $\mathcal{F}$:

$$\phi : \mathcal{X} \rightarrow \mathcal{F}, \tag{4.5}$$

$$\psi : \mathcal{F} \rightarrow \mathcal{X}. \tag{4.6}$$

The training process of the Autoencoder is determined as minimizing the loss function $L(x, \psi(\phi(x)))$ that penalizes dissimilarity between the input and reconstructed output, such as the mean squared error [32].

Most typical architecture of the Autoencoder is undercomplete, meaning the dimensionality of a latent space is significantly smaller than that of the input space. During the training process, the model is "forced" to learn the most salient features of the input data, i.e. regular patterns, generalizing over the train set. By definition anomalies are considered an irregularity, therefore undercomplete Autoencoders fail to reconstruct anomalous data with the same level of precision as compared to normal data.

Generally, an Autoencoder for Anomaly Detection can be built using any type of neural network layers (i.e. feed-forward linear layers [30], convolutional layers [33], etc.). Because this work is focused on the Anomaly Detection performed on the time-series data, it is profitable to use recurrent data behavior in order to reach higher performance.

Unlike previously published undercomplete autoencoder [5] used for the same application, in the present work we use a different approach. The implemented model has no regular bottleneck, the latent space $\mathcal{F}$ has the same dimensionality as the input $\mathcal{X}$, at the same time the information loss is achieved by using the Dropout layer on the output of encoder ϕ, which randomly sets input units to 0 with given *dropout rate*. Another Dropout layer is then applied to the output of decoder ψ, before the final reconstruction. Such architecture is mostly similar to Sparse Autoencoder [34].

We implemented two types of Autoencoders using recurrent neural network layers: Long Short-Term Memory (LSTM) cells and Gated Recurrent Unit (GRU) cells.

4.2.1 LSTM Autoencoder

Long Short-Term Memory (LSTM) Autoencoder uses layers composed of LSTM cells [35] in the encoder-decoder architecture.

LSTM Cell

LSTM cell is a type of gated Recurrent Neural Network (RNN) that allows learning the information selectively, i.e. retaining "important" features, and forgetting "less important". It consists of an input gate, memory cell, forget gate, and output gate, as shown in Figure 4.2.

The forget gate decides if the network should keep the information from the previous time step or forget it. This is decided by forget vector f_t, which is obtained by passing the current state X_t and previously hidden state h_{t-1} into the first sigmoid function.

The input gate performs two operations to update the cell state. First, the current state X_t and previously hidden state h_{t-1} are passed into the second sigmoid function to produce i_t which quantifies the importance of the new information carried by the current state, ranging from 0 (important) and 1 (not important). Second, the same information of the hidden state h_{t-1} and current state X_t are passed through the tanh function, which creates a vector $\widetilde{C}_t$ with values ranging from -1 to 1. The output values i_t and $\widetilde{C}_t$ generated from the activation functions are then multiplied forming the output of the Input gate.

The previous cell state C_{t-1} gets multiplied with forget vector f_t. If the outcome is 0, then values will get dropped from the cell state. The network then takes this output value and output of the Input gate and performs addition, which updates the cell state giving the network a new cell state C_t.

The output gate decides the value of the new hidden state h_t. This state contains information about previous inputs. First, the values of the current state X_t and previous hidden state h_{t-1} are passed into the third sigmoid function. Then the new cell state C_t is passed through the tanh function. Both these outputs are then multiplied. Based on the final value, the network decides which information the hidden state h_t should carry. This hidden state is used for prediction. Finally, the new cell state C_t and new hidden state h_t are carried over to the next time step [3, 36].

Implementation

LSTM Autoencoder is implemented using TensorFlow and the model is summarized in Algorithm 4.

The first LSTM layer with *n_units* processes the input window $W_{i,k}$ in a sequential way, one metric with L time steps at a time, which corresponds to processing the window one row at a time. The output forms a matrix $X_1 \in \mathbb{R}^{n_units \times L}$ (line 2).

Matrix X_1 is then passed through the Dropout layer, which randomly sets input units to 0 with a ratio of *dropout_rate* forming matrix X_2 (line 3).

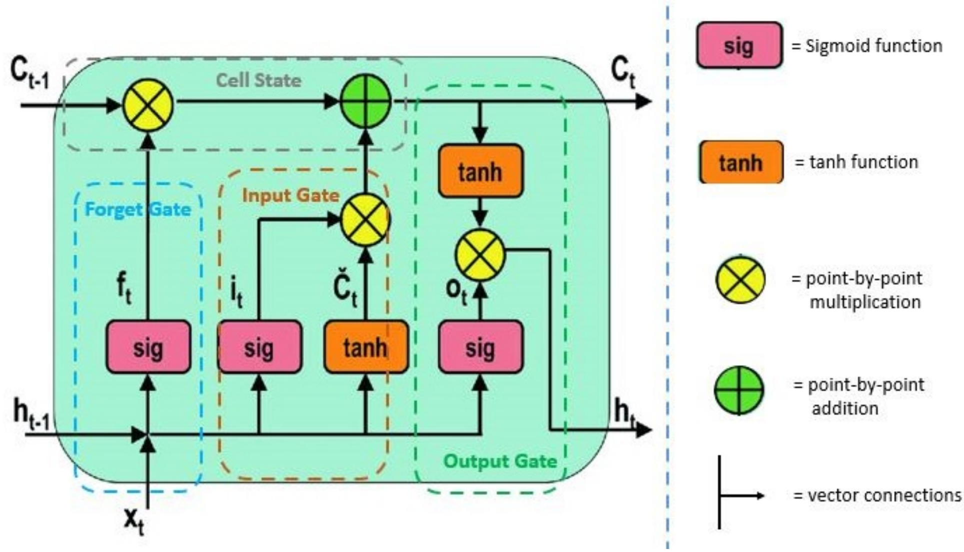


Figure 4.2: LSTM Cell [3]

Another LSTM layer with n_units processes the previous output X_2 in the same way as first LSTM layer, creating a matrix $X_3 \in \mathbb{R}^{n_units \times L}$ (line 4).

Matrix X_3 is then passed through another Dropout layer with the same $dropout_rate$ (line 5).

Finally, K Dense output layers are applied to each metric matrix X_4 independently, which correspond to the rows of X_3, obtaining a final reconstructed window $\widetilde{W} \in \mathbb{R}^{K \times L}$ (line 6).

In the end, the Anomaly score is computed by the $DIST$ function (line 7).

Algorithm 4: LSTM-Autoencoder

Data: $W_{i,k}$

Result: Anomaly score for the window $W_{i,k}$

1. $W = W_{i,k}$
2. $X_1 = LSTM(n_units)(W)$
3. $X_2 = DROPOUT(dropout_rate)(X_1)$
4. $X_3 = LSTM(n_units)(X_2)$
5. $X_4 = DROPOUT(dropout_rate)(X_3)$
6. $\widetilde{W} = TIME_DISTR(DENSE(n_neurons = K))(X_4)$
7. $DIST(W, \widetilde{W})$

Training Phase

In the beginning, the whole training set TR undergoes another normalization. For each of the K metrics, min and max values are obtained across the whole set. These values are then used for the $min-max$ normalization of metrics in all windows resulting in values ranging from 0 to 1. Vectors min and max of length K are then stored as part of the model and later used in the Inference Phase.

Then the network is trained on the reconstruction objective with a mean squared error loss, using the Adam optimizer.

Inference Phase

Each metric window under study $W_{i,k}$ undergoes $min - max$ normalization as described in Training Phase.

Then the network processes the normalized window and produces a reconstructed window $\widetilde{W}_{i,k}$. The final anomaly score is the mean squared error of the reconstructed window, which can also be written as:

$$\mathcal{A}(W_{i,k}) = ||W_{i,k} - \widetilde{W}_{i,k}||_F, \tag{4.7}$$

where $||X||_F$ denotes the Frobenius norm of a matrix, namely the ℓ_2 norm of the vector $\vec{x}$ of the unfolded rows. This function is implemented by PyOD *pairwise_distances_no_broadcast*.

4.2.2 GRU Autoencoder

Gated Recurrent Unit (GRU) Autoencoder uses layers composed of GRU cells [37] in the encoder-decoder architecture.

GRU Cell

GRU is an artificial recurrent neural network (RNN) that shares many properties with LSTM. Both algorithms use gating mechanisms to control the memorization process. GRU is less complex than LSTM and is significantly faster to compute because it combines the Input and Forget gate into a single Update gate. GRU consists of an Update gate and Reset gate, as shown in Figure 4.3.

Even though GRU has simpler architecture it exposes the complete memory (as opposed to LSTM). This allows the GRU-based models to often outperform LSTM-based models while being computationally simpler [38, 39].

The reset gate is responsible for the short-term memory of the network. It decides if information from the previous time step is kept or forgotten. First it adds the current state X_t with previous hidden state h_{t-1}. Second, it passes the sum through the second sigmoid function forming reset vector r_t ranging from 0 to 1.

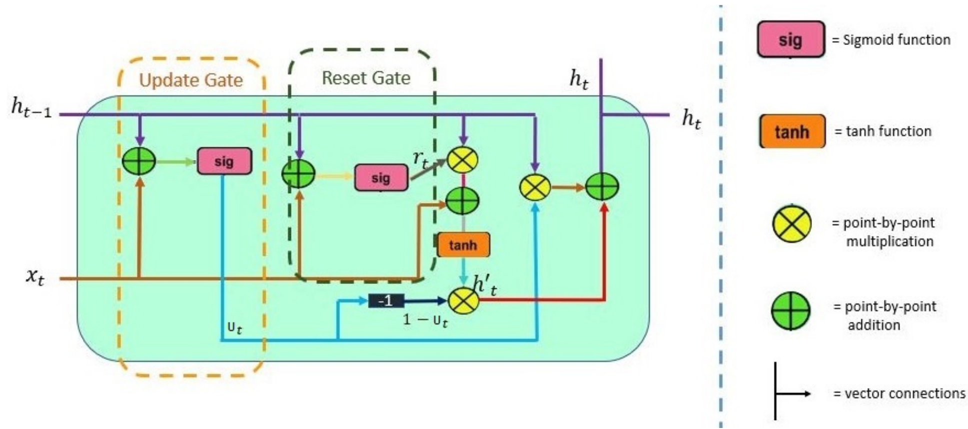


Figure 4.3: GRU Cell [4]

Reset vector r_t is then multiplied by previous hidden state h_{t-1} , then product is added to the current state X_t and passed through tanh function forming hidden state candidate h'_t.

Update gate forms update vector u_t the same way as Reset gate does with reset vector r_t – first, it does the addition of the current state X_t and previous hidden state h_{t-1} and passes the sum through sigmoid function.

In the production of final hidden state h_t update vector u_t comes into the picture. It will determine which information to collect from hidden state candidate h'_t and previous hidden state h_{t-1}.

Multiplication is applied to the update vector u_t and previous hidden state h_{t-1}. This product is then added to the multiplication of $1 - u_t$ and hidden state candidate h'_t. This sum forms the final hidden state h_t [4, 40].

Implementation

GRU-AE utilizes the same implementation as described in Section 4.2.1, replacing LSTM layers with GRU layers.

4.3 Ensemble

Different machine learning models are able to learn different patterns characterizing normal windows. Based on this we decided to aggregate the three anomaly score functions $\mathcal{A}_\lambda$ in an ensemble.

We define the detection output of each ensemble (ENS_e) by aggregating the individual outcomes through voting strategies, to overcome issues arising from different ranges of anomaly scores produced by $\mathcal{A}_\lambda$ of each model, i.e. by counting how many individuals trigger a detection, thus satisfy:

$AD_\lambda(W_{i,k}, \Gamma_\lambda) = 1$ for a given metric window $W_{i,k}$:

$$ENS_e(W_{i,k}) = \sum_\lambda AD_\lambda(W_{i,k}, \Gamma_\lambda) \geq e, e = 1, 2, 3. \tag{4.8}$$

These ensembles correspond to the implementation of the voting strategies "OR", "MAJ", and "AND" for $e = 1, 2, 3$ respectively.

CHAPTER **5**

Labeled Dataset

In this chapter, we describe the labeled dataset and its preparation.

The main use of the labeled dataset is to benchmark the Anomaly Detection methods under study for the CERN use case. This benchmark gives a fast and objective way to evaluate the Anomaly Detection System performance compared to qualitative feedback from service managers after months of extensive usage of our proposed Anomaly Detection System.

The dataset is based on a *shared* hostgroup, which contains 40 hypervisors. We have annotated 2 months of data from September 1st to October 31st 2021. The choice of this specific HG was based on the relatively small number of HVs (some *shared* HGs contain around 200 HVs) and a reasonable number of anomalous samples, where some of them could have been spotted at the first glance.

The annotation process was mainly done by 2 independent annotators and guided by the CERN cloud service managers that have shared their experience to identify critical anomalies and exclude harmless behaviors not affecting the cloud service level objectives. In case of disagreement between the annotators, the final decision included the opinion of additional expert service managers.

The final dataset contains labels for 32 HVs and is composed of 11712 4-hours long time windows, from which 228 are anomalous and 11484 are normal. The percentage of anomalies in the whole dataset is therefore equal to 2%. Figure 5.1 shows a visualization of the dataset.

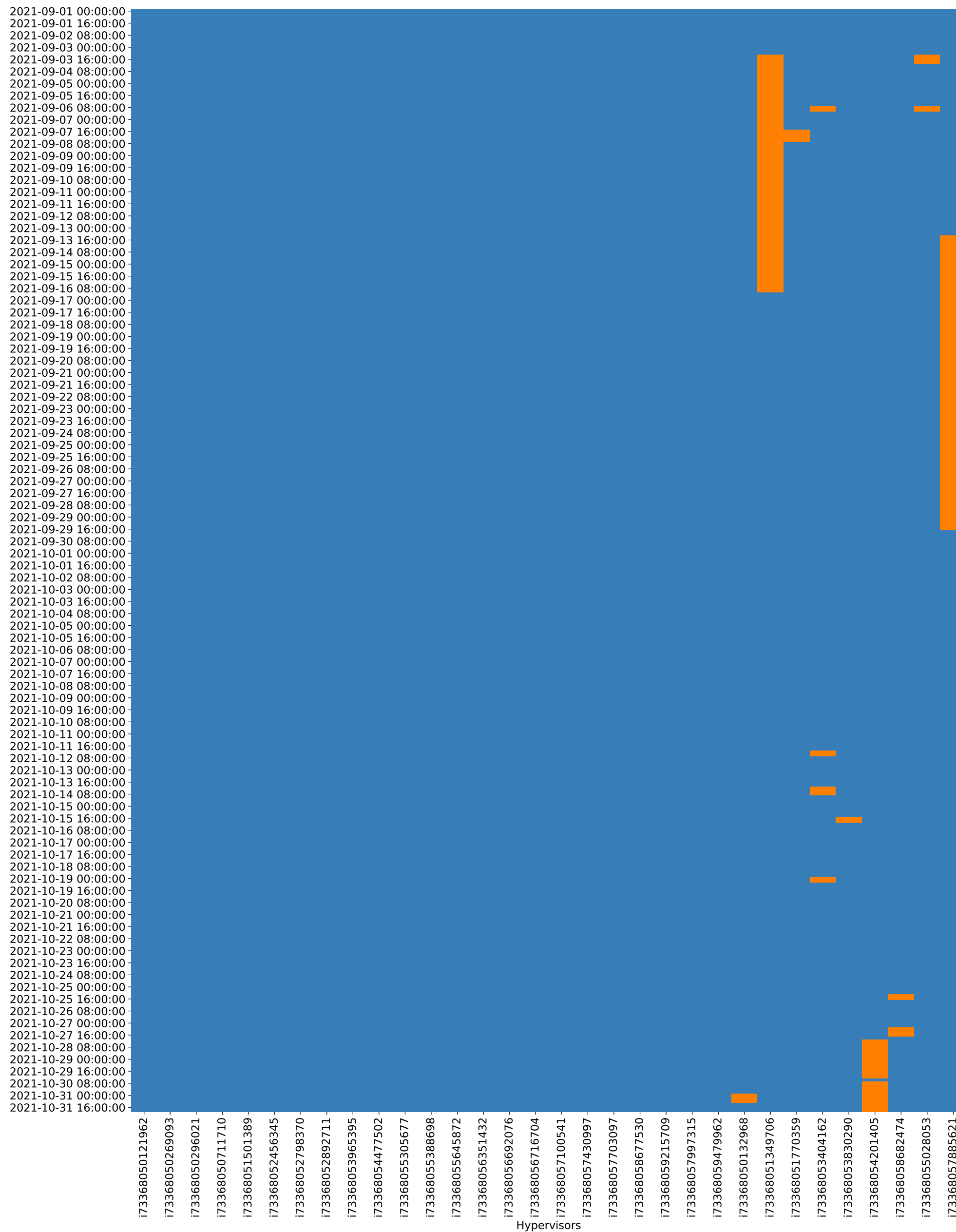


Figure 5.1: Labeled Dataset based on a Shared Hostgroup: blue = normal, orange = anomaly

CHAPTER 6

Experiments and Results

In this chapter, we will:

- introduce used performance metrics,
- define the figure of merit,
- evaluate the performance of individual models,
- discuss the choice of parameters,
- introduce the Current Alarming System,
- compare the Current Alarming System with respect to our proposed solution.

We present two different sets used for the training of the models and inference of the scores:

- first approach is to train the models on a single HG of *gva_shared_019* containing 40 HVs and infer anomaly scores for the same set of HVs,
- second approach is to train the models on the whole cloud, meaning multiple HGs *gva_shared_** containing 1700 HVs in total and infer anomaly scores for the same set of HVs.

All of the experiments are done using the first approach (if not stated otherwise) with metrics introduced in 6.2, evaluated on the labeled dataset presented in Chapter 5, and based on time windows of 4 hours with an *aggregation period* of 30 minutes.

6.1 Used Resources

All experiments have been done on a Centos 7 VM in the CERN OpenStack cloud, which consists of 16 CPU cores, 32GB of RAM, 160GB of local storage, and access to EOS distributed storage.

The somewhat surprising paradox is that the HV hosting this VM has been flagged as anomalous multiple times by the AD System (itself), due to heavy load during excessive experiments.

6.2 Input metrics

Based on the expert experience, previous work [5], and our previous experiments, we have chosen 6 performance metrics, which are described in Table 6.1 including the thresholds used by the Current Alarming system, which will be introduced later (Sec. 6.5).

Metric's name	Description	Threshold of the current system
Context Switches	Frequency of involuntary context switches, which occur when a process consumes more CPU time than what it was allocated by the kernel	> 50 kHz
CPU Load	Number of processes using CPU, waiting to use CPU, or waiting for input/output access averaged over 5 minutes divided by number of CPU cores	> 2
CPU System	Percentage of CPU time used by the kernel	$> 35\%$
Disk IO Time	Portion of time spent every 1 second doing input/output operations	> 900 ms
Disk Pending Operations	Queue size of pending input/output operations	> 25
Memory Free	Memory not being used	< 500 MB

Table 6.1: Monitoring metrics used by the proposed solution and used by the current alerting system.

6.3 Figure of Merit

All of the models under study produce an anomaly score. By applying a threshold as explained in 4.1models label a time window in two possible classes – anomalous or normal. We define the following four metrics:

- True Positives (TP): number of time windows labeled as anomalous in the dataset that the algorithm correctly identifies as anomalies,
- True Negatives (TN): number of time windows labeled as normal in the dataset that the algorithm correctly identifies as normal,
- False Positives (FP): number of time windows labeled as normal in the dataset that the algorithm incorrectly identifies as anomalies,
- False Negatives (TN): number of time windows labeled as anomalous in the dataset that the algorithm incorrectly identifies as normal.

Those four metrics form a *Confusion Matrix*, which is shown in Figure 6.1

Figure 6.1: Confusion Matrix

6.3.1 AUC-ROC

Area Under the Curve of Receiver Operating Characteristic (AUC-ROC) is a performance measurement for binary [4] classification problems using various threshold settings. ROC is a probability curve and AUC represents the degree of separability. It describes the model's capacity of dividing between the two classes. The higher the AUC, the better the model is at predicting normal samples as normal, and vice versa.

[4] it can be also used for multi-class classifications

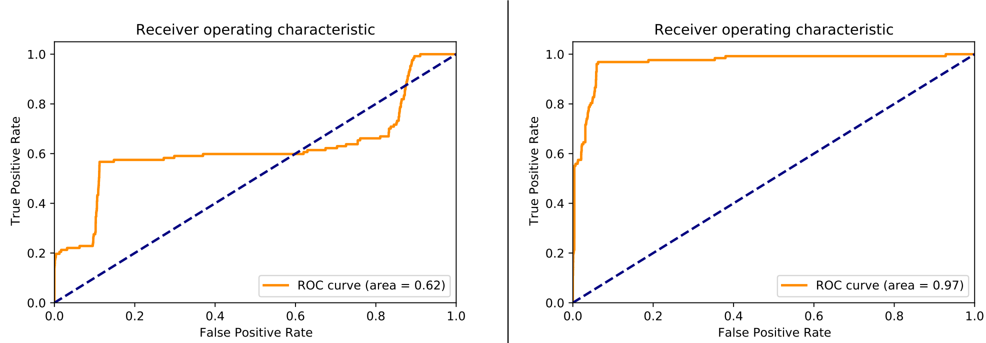


Figure 6.2: ROC and AUC of poorly performing model (left) and well-performing model (right)

ROC is based on the True Positive Rate (TPR) and False Positive Rate (FPR):

$$TPR = \frac{TP}{TP + FN} = \frac{TP}{P}, \tag{6.1}$$

$$FPR = \frac{FP}{FP + TN} = \frac{FP}{N}, \tag{6.2}$$

where N denotes the total number of *normal* samples in the dataset and P the total number of *anomalous* samples.

The ROC curve is plotted with TPR against the FPR where TPR is on the y-axis and FPR is on the x-axis for multiple threshold values of the anomaly scores. Each point in the plot of ROC corresponds to a threshold of a different value.

A great model has an AUC near 1 – it is capable of separating the scores very well. A model with an AUC near 0 has a very poor measure of separability. In fact, model with an AUC of 0 is inverting the classes. A model with an AUC of 0.5 does not have any separation capacity (random binary classifier). Figure 6.2 shows an example of the ROC and AUC for both poorly and well-performing models.

The main reason for using the AUC-ROC in our case is that we can compare models without the need to explicitly set a threshold, but we can compare their performance with various threshold values. This performance measurement was also used in the previous work [5].

6.3.2 Training and Inference time

Another important aspect of a model is its training and inference time. Note that, in our case, the time of training is not so critical due to the re-use of

saved models.

6.4 Performance of the Individual Models

In this section, we present experiments to compare the performance of individual models with different parameters and periods of the training.

6.4.1 Isolation Forest

We use IFOR with the default parameters introduced by PyOD. The only changed parameter is the number of trees, which has a default value of 100.

Experiments with different numbers of trees are presented in Table 6.2. Results are averaged over 12 experiments – 4 different training periods of 1 week using 3 different random seeds to initialize the model. Even though there is not a significant improvement, we decided to **use 200 iTrees as the final value** due to the highest AUC-ROC and lowest standard deviation making the model more stable (less dependent on the choice of the training week and random seed).

Num. of iTrees	AUC-ROC	Train time (s)	Infer time (s)
50	0.953 ± 0.007	0.85 ± 0.48	59.9 ± 15.4
100	0.955 ± 0.007	0.88 ± 0.04	68.4 ± 12.4
150	0.956 ± 0.005	1.18 ± 0.05	87.1± 16.9
200	0.957 ± 0.004	1.56 ± 0.09	100.3 ± 19.6

Table 6.2: AUC-ROC, training and inference time of IFOR with different numbers of iTrees

Another experiment to test the effect of different training periods is presented in Table 6.3. Two lengths of training periods are evaluated – 1 week and 1 month. Results for 1 week are averaged over 45 experiments – 15 different training periods of 1 week using 3 different seeds. Results for 1 month are averaged over 12 experiments – 4 different training periods of 1 month using 3 different seeds. A longer training period provides a little improvement regarding the AUC-ROC with a lower value of standard deviation compared to the shorter period. Even though a week of training is enough for the model (based on the AUC-ROC), it further benefits from the longer training period of a month.

Training Length	AUC-ROC	Train time (s)	Infer time (s)
Week	0.954 ± 0.006	1.64 ± 0.29	94.9 ± 16.8
Month	0.959 ± 0.004	2.91 ± 0.05	95.1 ± 13.1

Table 6.3: AUC-ROC, training and inference time of IFOR with different lengths of the training period

6.4.2 Autoencoders

In this section, we evaluate the proposed architecture for Autoencoders. Experiments are based on using different parameters (namely *dropout rate* and *the number of units* as described in Section 4.2.1), different training periods, reasons for some of the architectural decisions, and comparison with architecture proposed in previous work [5].

For all of the experiments models were trained for 50 epochs without early stopping, as we found this approach the most suitable for the unsupervised technique. Based on the observations of training/validation loss, most models reached a plateau around 30 to 40 epochs in.

Choice of Parameters

In earlier experiments with a different set of input metrics dropout rates of the following values were observed: 0, 0.1, 0.2, 0.3. The best results were obtained for dropout rates of 0.2 and 0.3 and the final **chosen value of dropout rate is 0.25**. Due to time limitations, experiments of the same scale regarding dropout rate on Input metrics presented in 6.2 won't be included in this work.

Experiments with a different number of units and 2 values of dropout rate are presented in Table 6.4 for the LSTM-AE and in Table 6.5 for the GRU-AE. Results are averaged over 12 experiments – 4 different training periods of 1 week using 3 different seeds.

Parameters	AUC-ROC	Train time (s)	Infer time (s)
1 Unit, 0.25 Dropout	0.971 ± 0.004	62.3 ± 9.1	114.2 ± 13.1
1 Unit, 0 Dropout	0.917 ± 0.045	63.8 ± 3.1	105.4 ± 17.6
3 Units, 0.25 Dropout	0.956 ± 0.016	64.5 ± 6.2	95.3 ± 14.1
3 Units, 0 Dropout	0.892 ± 0.026	69.2 ± 5.3	113.8 ± 24.9
5 Units, 0.25 Dropout	0.960 ± 0.007	64.9 ± 2.6	100.3 ± 17.1
5 Units, 0 Dropout	0.877 ± 0.034	63.9 ± 6.3	113.3 ± 16.2

Table 6.4: AUC-ROC, training, and inference time of LSTM-AE with different parameters

Parameters	AUC-ROC	Train time (s)	Infer time (s)
1 Unit, 0.25 Dropout	0.967 ± 0.007	81.6 ± 11.9	111.9 ± 29.1
1 Unit, 0 Dropout	0.859 ± 0.097	109.9 ± 17.6	89.5 ± 10.0
3 Units, 0.25 Dropout	0.964 ± 0.007	76.6 ± 7.1	102.2 ± 17.7
3 Units, 0 Dropout	0.794 ± 0.099	80.8 ± 7.2	98.4 ± 17.5
5 Units, 0.25 Dropout	0.961 ± 0.008	76.3 ± 5.6	94.5 ± 11.9
5 Units, 0 Dropout	0.805 ± 0.079	79.9 ± 3.8	98.8 ± 15.9

Table 6.5: AUC-ROC, training, and inference time of GRU-AE with different parameters

Figure 6.3 shows training and validation loss of the worst-performing GRU model (5 Units, 0 Dropout) with an AUC-ROC of 0.617. Figure 6.4 shows the same for the well-performing GRU model (1 Unit, 0.25 Dropout) on the same training week and random seed with an AUC-ROC of 0.967. With a Dropout rate of 0 models learn to mimic the input data too precisely (demonstrated by Model loss in Fig 6.3), which makes the detection of anomalies based on reconstruction error unusable since the model is not able to separate the scores (as shown by Distribution of scores in Fig. 6.3).

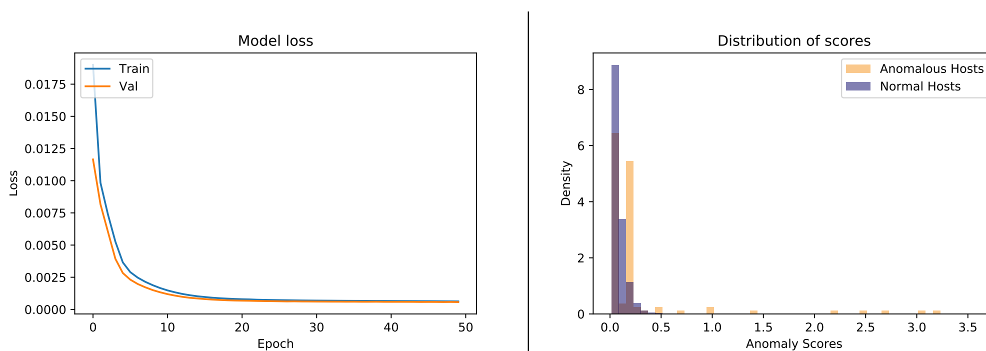


Figure 6.3: Training/validation loss and distribution of the scores of poorly performing GRU model (5 Units, 0 Dropout) with AUC-ROC of 0.617

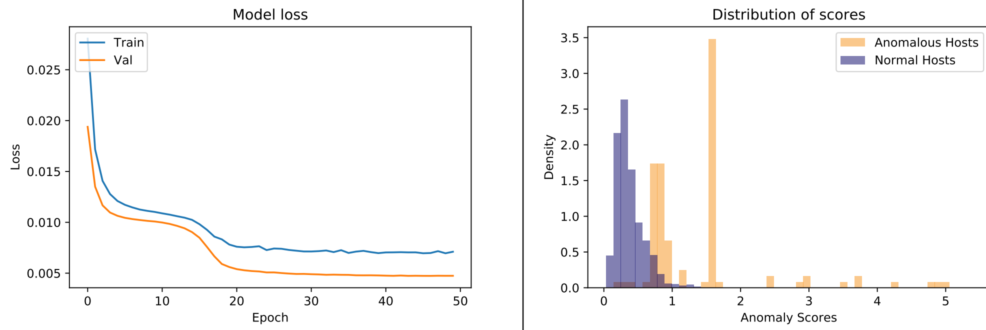


Figure 6.4: Training/validation loss and distribution of the scores of well-performing GRU model (1 Unit, 0.25 Dropout) with AUC-ROC of 0.967

All experiments with the dropout rate of 0.25 show very similar results regarding the AUC-ROC. However, 1 Unit performs the best with the lowest value of standard deviation while keeping the complexity of the model lower compared to the higher number of units. Therefore it is chosen as the final value.

Note that in experiments with the dropout rate of 0 we can no longer label the models as Autoencoders since there is no bottleneck in any form meaning no encoding/decoding is happening. However, the presented experiments are to demonstrate why we decided to use dropout in the final architecture.

Training Period

In this experiment, we test the effect of different training periods with parameters of **1 Unit and 0.25 Dropout rate**. Two lengths of training periods are evaluated – 1 week and 1 month. Results for 1 week are averaged over 45 experiments – 15 different training periods of 1 week using 3 different seeds. Results for 1 month are averaged over 12 experiments – 4 different training periods of 1 month using 3 different seeds.

Table 6.6 shows the results for LSTM-AE, and Table 6.7 for GRU-AE. Based on the results LSTM-AE does not benefit from the longer training period since the AUC-ROC and its standard deviation are exactly the same. On the other hand, the better performance of the GRU-AE in terms of AUC-ROC and its standard deviation with a longer period of training shows GRU-AE benefits from more training samples compared to LSTM-AE.

Training Length	AUC-ROC	Train time (s)	Infer time (s)
Week	0.964 ± 0.010	64.6 ± 9.6	107.8 ± 19.4
Month	0.964 ± 0.010	106.1 ± 6.9	103.6 ± 15.5

Table 6.6: AUC-ROC, training and inference time of LSTM-AE with different lengths of the training period

Training Length	AUC-ROC	Train time (s)	Infer time (s)
Week	0.962 ± 0.010	83.2 ± 9.8	100.1 ± 22.8
Month	0.971 ± 0.005	146.4 ± 26.5	97.8 ± 13.6

Table 6.7: AUC-ROC, training and inference time of GRU-AE with different lengths of the training period

Architectural Decisions

Usage of the 2 Dropout Layers (as we present in Sec. 4.2.1) might be unusual, however in this experiment, where the second Dropout Layer is not present, we show a reason for doing so. Table 6.8 presents results averaged over 12 experiments – 4 different training periods of 1 week and 3 different seeds. The choice of parameters is the same as in previous experiments – 1 Unit, 0.25 Dropout rate. Compared to the "Week" in Table 6.6 and Table 6.7 we see both models benefit in terms of average AUC-ROC and its standard deviation from a stronger form of regularization achieved by using 2 Dropout layers, especially the GRU-AE.

Model	AUC-ROC
LSTM-AE	0.949 ± 0.016
GRU-AE	0.925 ± 0.033

Table 6.8: AUC-ROC of LSTM-AE and GRU-AE with a single Dropout layer

Comparison with previous architecture

As mentioned before, previous work presented an Undercomplete LSTM-AE [5]. In this experiment, we compare its performance with our proposed architecture of LSTM-AE using selected parameters. Table 6.9 displays results based on 3 different values of Units. Each of the results is averaged over 12 experiments – 4 different training periods of 1 week and 3 different seeds.

Compared to Table 6.6 we show that our proposed architecture significantly outperforms the previous architecture in terms of AUC-ROC and its standard deviation in all cases.

Parameters	AUC-ROC
1 Unit	0.877 ± 0.059
3 Units	0.839 ± 0.047
5 Units	0.812 ± 0.039

Table 6.9: AUC-ROC of LSTM-AE proposed in previous work [5]

6.4.3 Summary

To summarize the results of individual models, we present Table 6.10 which shows the performance of all 3 models (IFOR, LSTM-AE, GRU-AE) for 2 lengths of training periods. The final chosen parameters are 200 iTrees in the case of IFOR, and 1 Unit with a 0.25 Dropout rate for both LSTM-AE and GRU-AE.

Model	AUC-ROC Week	AUC-ROC Month
IFOR	0.954 ± 0.006	0.959 ± 0.004
LSTM-AE	0.964 ± 0.010	0.964 ± 0.010
GRU-AE	0.962 ± 0.010	0.971 ± 0.005

Table 6.10: Stability of model performance based on the average and standard deviation of the AUC-ROC for one week-long (AUC-ROC week) and one month-long (AUC-ROC Month) training periods

Each of the proposed models performs very well in terms of AUC-ROC and the low value of its standard deviation shows that the choice of specific training period does not play a big role. This confirms that the presented models are robust.

In our use case, IFOR performs on the same level of performance in terms of AUC-ROC as both LSTM-AE and GRU-AE with much simpler architecture and requires a small amount of time to train. On the other hand, inference time is on the same scale as for Autoencoders. If training time and simple architecture were a big concern for us, IFOR alone would be a great solution.

Even though GRU-AE should be significantly faster than LSTM-AE in the training and inference due to simpler architecture, in the case of TensorFlow layers implementation this does not happen. Comparing Table 6.6 and Table 6.7 we see GRU-AE is slower in the training and insignificantly faster in inference.

6.5 Comparison with Current Alarming System

The main goal of this work is to develop an AD system, which outperforms the Current Alarming System based on static threshold alarming implemented in the CERN cloud. In this section, we introduce the Current Alarming System and compare its performance with respect to our proposed AD system.

6.5.1 Current Alarming System

The current Alarming System is based on collectd metrics and Grafana alerts. It is composed of 6 performance metrics selected by service managers. The metrics were selected because they are popular indicators for the computing performance of HVs and are often used in post-mortem analysis. Table 6.1 displays a short description of each metric with its critical value (threshold) from which the system starts to notify about anomalies. This system examines each monitoring metric independently from the others and triggers alarms when any metric of any HV exceeds its threshold value. In order to reduce the alarming rate due to fluctuating metrics that temporarily exceed the threshold, the metrics are analyzed in windows of 12 hours. The service managers consider this potential delay of reaction caused by the longer window time resolution as a reasonable trade-off between effectively identifying real operational issues and the number of false positives. This approach is built on the fact that computing systems in large data centres use a number of self-healing strategies to recover from issues. Figure 6.5 shows an example of Grafana alerts in the Current Alarming System for the CPU Load metric.

In the current system, the alarm thresholds are set very high compared to the normal metric distribution of a cluster to achieve a very low value of FPR. It should be noticed that in a data centre of several thousands of servers, a FPR of just 1% would result in tens of servers being uselessly inspected every day.

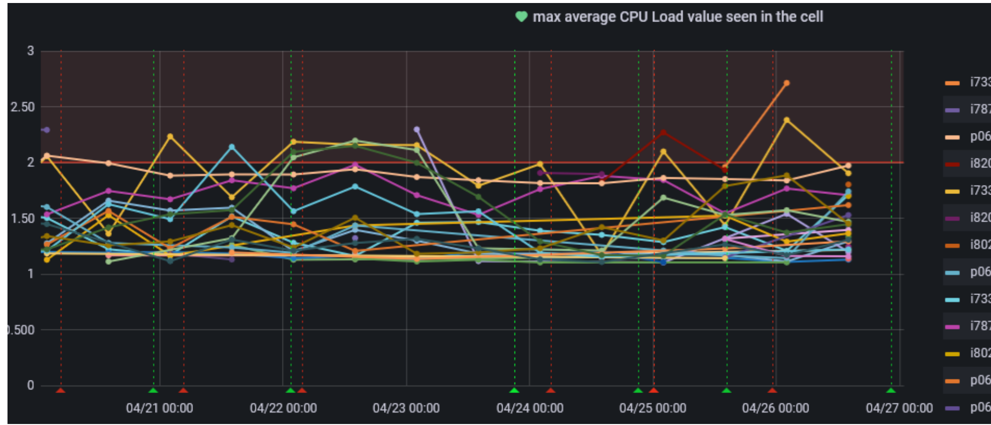


Figure 6.5: CPU Load alerts in the Current Alarming System: horizontal red line = threshold, vertical red arrow = alert triggered, vertical green arrow = previous alert no longer triggered

6.5.2 Evaluation and Comparison

In this experiment, we compare the performance of our (individual and ensemble) models with respect to the performance of the Current Alarming System. Our models were trained on the whole cloud (1700 HVs) with a training period of 1 month.

A fair comparison is guaranteed by configuring the current system to analyze each metric in windows of 4 hours (instead of 12 hours). For each window of each metric, we compute the average value and compare it against its threshold. Thresholds are manually adjusted to achieve different target values of FPR in the following experiments. The predictions of each model were evaluated on the labeled dataset (Chap. 5), to compute the TPR and FPR. The performance is compared on the achieved TPR value at a given value of FPR. For each model, we have identified the value of the threshold (in the case of Ensemble multiple thresholds) Γ_λ that produce the same value of FPR. The same approach is also applied to the alarming thresholds of the current system.

In Table 6.11 the TPR values are reported for all approaches, evaluated at four different values of FPR, ranging from 0.1% to 4%. The reason for so small values of FPR has been already discussed: given a large number of servers in the cloud infrastructure, a small FPR is mandatory to avoid that false notifications overwhelm the service managers. The FPR of the current system is 0.1% based on the evaluation on the labeled dataset.

All of our proposed models outperform the Current Alarming System in terms of achieved TPR for any given FPR. Between the individual models, the Autoencoders have the best performance on the whole range of studied

FPR	Current System	True Positive Rates					
		Individual			Ensemble		
		IFOR	LSTM-AE	GRU-AE	ENS_1	ENS_2	ENS_3
0.001	0.08	0.09	0.45	0.43	0.47	0.45	0.21
0.01	0.14	0.29	0.56	0.58	0.53	0.58	0.59
0.02	0.19	0.57	0.66	0.79	0.61	0.69	0.71
0.04	0.26	0.81	0.81	0.93	0.92	0.89	0.92

Table 6.11: True positive rate (TPR) measured at different values of the False Positive Rate (FPR) for the predictions of the Current Alarming System, the proposed individual models, and the proposed ensemble methods ENS_e for $e = 1, 2, 3$

FPR, with the GRU-AE that achieves the largest TPR value for 4% of FPR. The performance of the IFOR at the lowest FPR is comparable with the one of the current system, whereas it becomes comparable with the LSTM-AE at 4% of FPR.

ENS_1 and ENS_2 give the best performance at the lowest FPR, but the inefficacy of IFOR penalizes ENS_3. At the largest considered FPR (4%), the three ensemble methods have similar performance. We remind that the three ensemble methods ENS_e are implemented with the voting strategies ”OR”, ”MAJ” and ”AND” for $e = 1, 2, 3$ respectively (Sec. 4.3).

Based on the presented results, our proposed models have been integrated into the data analytic infrastructure that processes the CERN cloud monitoring data. The predictions of the ensemble method ENS_3 are now part of the Grafana dashboards used by the cloud service managers to identify disturbing events (Fig. 6.6).

We regard as a great achievement the satisfactory experience reported by the service managers, which also qualitatively confirms the effectiveness of our solution.

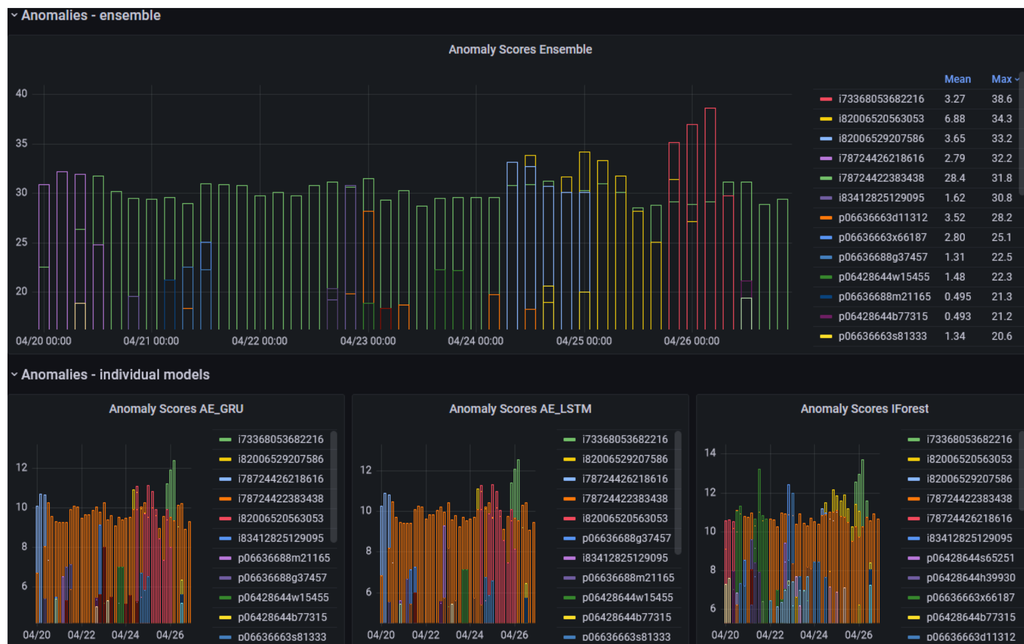


Figure 6.6: Grafana Dashboard containing anomaly scores from ensemble method ENS_3 and individual models for the time period of 7 days

Chapter 7

Future Work

The biggest downside of this work is the absence of online data. It has been discussed with the MONIT team that in the future they could offer an API that can offer unaggregated data for ”any” time period. This could be achieved by reading the raw collectd data for $T-1$ days from HDFS and online data directly from Kafka, or by querying specific InfluxDB backends based on the defined metrics. If this feature is not provided, another option could be change in the policy of how the HDFS exposes data – data for $T-1$ are available to the users, but newer data remain inaccessible to users until the next day when the data undergo compaction. If this is also not an option, the support for querying multiple storage backends has to be implemented in our libraries.

When access to online data is provided, the next logical step is to do research about forecasting, to predict potential future anomalies.

The last future goal is to completely switch to the proposed Machine Learning Anomaly Detection System replacing the Current Alarming System and integrating it into the Daily Operations of all service managers.

Conclusion

The main goal of this work was to develop an Anomaly Detection System, which outperforms the Current Alarming System based on static threshold alarming implemented in the CERN cloud. We modified and extended the existing Proof of Concept [5], implemented new algorithms, and created a labeled dataset. The proposed solution, composed of Isolation Forest, LSTM Autoencoder, GRU Autoencoder, and Ensemble methods, outperforms the Current Alarming System in terms of obtained True Positive Rate for any given False Positive Rate. The performance of our solution is further confirmed by the satisfactory experience reported by the service managers. This solution was integrated into the data analytic infrastructure that processes the CERN cloud monitoring data.

In Chapter 1 we described the structure and used technologies of CERN Monitoring Infrastructure. In Chapter 2 we defined the important terms for the scope of this work and introduced the problem formulation. In Chapter 3 we described the Architecture of the Anomaly Detection Pipeline with the description of individual steps and explained how the Productional Deployment is done. In Chapter 4 we introduced the individual models and defined the ensemble methods. In Chapter 5 we emphasize the importance of a labeled dataset, how we created it, and statistics about the dataset. In Chapter 6 we did experiments and evaluations regarding the performance in terms of AUC-ROC for all 3 of the individual models, the ensemble, and comparison of the Proposed solution with respect to the Current Alarming System based on the obtained True Positive Rate with a given False Positive Rate.

Results of this work were presented at the (Virtual) Computing in High Energy Physics 2021 (vCHEP2021) conference on track of Networks and facilities [5] and on HEPiX Spring 2022 online Workshop on track of Computing & Batch Services, Grid, Cloud & Virtualisation [6].

[5] https://indico.cern.ch/event/948465/contributions/4323967

[6] https://indico.cern.ch/event/1123214/contributions/4809938

Bibliography

[1] Nikolay Tsvetkov. Building scalable & reliable monitoring system [online]. [cit. 2022-02-21]. Available from: https://indico.cern.ch/event/930896/contributions/3938294/attachments/2071885/3478355/2020-07-09_OPENLAB-MONIT.pdf

[2] Hariri, S.; Kind, M. C.; et al. Extended Isolation Forest. *IEEE Transactions on Knowledge and Data Engineering*, volume 33, no. 4, 2021: pp. 1479–1489, doi:10.1109/TKDE.2019.2947676.

[3] Singhal, G. Introduction to LSTM Units in RNN [online]. [cit. 2022-04-18]. Available from: https://www.pluralsight.com/guides/introduction-to-lstm-units-in-rnn

[4] Singhal, G. LSTM versus GRU Units in RNN [online]. [cit. 2022-04-18]. Available from: https://www.pluralsight.com/guides/lstm-versus-gru-units-in-rnn

[5] Paltenghi, M. Time Series Anomaly Detection for CERN Large-Scale Computing Infrastructure. Oct 2020, presented 02 Oct 2020. Available from: http://cds.cern.ch/record/2752641

[6] CERN. The matter-antimatter asymmetry problem [online]. [cit. 2022-02-17]. Available from: https://home.cern/science/physics/matter-antimatter-asymmetry-problem

[7] CERN. Welcome to the Worldwide LHC Computing Grid [online]. [cit. 2022-02-17]. Available from: https://wlcg.web.cern.ch/

[8] Openstack. What is OpenStack? [online]. [cit. 2022-02-19]. Available from: https://www.openstack.org/software/

[9] Red Hat. What is a hypervisor? [online]. [cit. 2022-02-17]. Available from: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor

[10] Red Hat. What is a virtual machine (VM)? [online]. [cit. 2022-02-17]. Available from: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine

[11] Openstack. How Ironic Delivers Abstraction and Automation using Open Source Infrastructure [online]. [cit. 2022-02-17]. Available from: https://www.openstack.org/use-cases/bare-metal/how-ironic-delivers-abstraction-and-automation-using-open-source-infrastructure

[12] Luis Fernández Álvarez. CERN Report: Batch farm worker nodes [online]. [cit. 2022-02-19]. Available from: https://indico.cern.ch/event/876806/contributions/4400263/attachments/2280883/3875388/CERN.pdf

[13] Aimar, A.; Corman, A.; et al. Unified Monitoring Architecture for IT and Grid Services. *Journal of Physics: Conference Series*, volume 898, 10 2017: p. 092033, doi:10.1088/1742-6596/898/9/092033.

[14] Nikolay Tsvetkov. Monitoring with no limits [online]. [cit. 2022-02-21]. Available from: https://twiki.cern.ch/twiki/pub/CMgroup/ExternalPresentationsIn2019/2019-10-07_MONIT_SPHERE_IT.pptx

[15] Aimar, A.; Corman, A.; et al. MONIT: Monitoring the CERN Data Centres and the WLCG Infrastructure. *EPJ Web of Conferences*, volume 214, 01 2019: p. 08031, doi:10.1051/epjconf/201921408031.

[16] Ariza Porras, C.; Kuznetsov, V.; et al. The CMS monitoring infrastructure and applications. *Computing and Software for Big Science*, volume 5, 12 2021, doi:10.1007/s41781-020-00051-x.

[17] collectd. collectd – The system statistics collection daemon [online]. [cit. 2022-02-25]. Available from: https://collectd.org/

[18] Apache Spark. Unified engine for large-scale data analytics [online]. [cit. 2022-02-27]. Available from: https://spark.apache.org/

[19] Hadoop. HDFS Architecture Guide [online]. [cit. 2022-02-27]. Available from: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[20] Antonín Dvořák. ELK Stack [online]. [cit. 2022-02-27]. Available from: https://cloudinfrastack.com/blog/monitoring/elk_stack/

[21] CERN. The Swan Service [online]. [cit. 2022-02-27]. Available from: https://swan.web.cern.ch/swan/

[22] Merriam-Webster. anomaly [online]. [cit. 2022-03-03]. Available from: https://www.merriam-webster.com/dictionary/anomaly

[23] Chandola, V.; Banerjee, A.; et al. Anomaly Detection: A Survey. *ACM Comput. Surv.*, volume 41, 07 2009, doi:10.1145/1541880.1541882.

[24] Berg, A.; Ahlberg, J.; et al. Unsupervised Learning of Anomaly Detection from Contaminated Image Data using Simultaneous Encoder Training. *ArXiv*, volume abs/1905.11034, 2019.

[25] ControlUp. What's a "Noisy Neighbor," and How Can ControlUp Shut Them Up? [online]. [cit. 2022-03-15]. Available from: https://www.controlup.com/resources/blog/entry/whats-a-noisy-neighbor-and-how-can-controlup-shut-them-up/

[26] Sai. Traditional and Representational Machine Learning [online]. [cit. 2022-04-18]. Available from: https://medium.com/@saiprakash513/traditional-and-representational-machine-learning-317495b74c1b

[27] Liu, F. T.; Ting, K.; et al. Isolation Forest. 01 2009, pp. 413 – 422, doi:10.1109/ICDM.2008.17.

[28] Preiss, B. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. 01 2000.

[29] MathWorks. What Is Deep Learning? [online]. [cit. 2022-04-18]. Available from: https://www.mathworks.com/discovery/deep-learning.html

[30] Sakurada, M.; Yairi, T. Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction. In *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, MLSDA'14, New York, NY, USA: Association for Computing Machinery, 2014, ISBN 9781450331593, p. 4–11, doi:10.1145/2689746.2689747. Available from: https://doi.org/10.1145/2689746.2689747

[31] Rumelhart, D. E.; Hinton, G. E.; et al. *Learning Internal Representations by Error Propagation*. Cambridge, MA, USA: MIT Press, 1986, ISBN 026268053X, p. 318–362.

[32] Goodfellow, I.; Bengio, Y.; et al. *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[33] Seyfioğlu, M.; Ozbayoglu, M.; et al. Deep Convolutional Autoencoder for Radar-Based Classification of Similar Aided and Unaided Human Activities. *IEEE Transactions on Aerospace and Electronic Systems*, volume PP, 02 2018: pp. 1–1, doi:10.1109/TAES.2018.2799758.

[34] Koike-Akino, T.; Wang, Y. Stochastic Bottleneck: Rateless Auto-Encoder for Flexible Dimensionality Reduction. 2020, doi:10.48550/ARXIV.2005.02870. Available from: https://arxiv.org/abs/2005.02870

[35] Hochreiter, S.; Schmidhuber, J. Long Short-term Memory. *Neural computation*, volume 9, 12 1997: pp. 1735–80, doi:10.1162/neco.1997.9.8.1735.

[36] shipra_saxena. Introduction to Long Short Term Memory (LSTM) [online]. [cit. 2022-04-18]. Available from: `https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm`

[37] Chung, J.; Gulcehre, C.; et al. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. 2014, doi:10.48550/ARXIV.1412.3555. Available from: `https://arxiv.org/abs/1412.3555`

[38] Yin, W.; Kann, K.; et al. Comparative Study of CNN and RNN for Natural Language Processing. 2017, doi:10.48550/ARXIV.1702.01923. Available from: `https://arxiv.org/abs/1702.01923`

[39] Jozefowicz, R.; Zaremba, W.; et al. An empirical exploration of recurrent network architectures. *Journal of Machine Learning Research*, 2015.

[40] shipra_saxena. Introduction to Gated Recurrent Unit (GRU) [online]. [cit. 2022-04-18]. Available from: `https://www.analyticsvidhya.com/blog/2021/03/introduction-to-gated-recurrent-unit-gru`

Acronyms

AD Anomaly Detection 12, 15, 17, 20

API Application Programming Interface 2, 49

AUC-ROC Area Under the Curve of Receiver Operating Characteristic 37

BST Binary Search Tree 24

CI/CD Continuous Integration / Continuous Delivery 21

CPU Central Processing Unit 2, 14

DAG Directed Acyclic Graph 21

FPR False Positive Rate 38

GRU-AE Gated Recurrent Unit Autoencoder 23, 40, 43

HDFS Hadoop Distributed File System 8

HG Hostgroup 11, 13, 16, 19, 35

HV Hypervisor 2, 11–14, 17, 19

IFOR Isolation Forest 23, 24

iTree Isolation Tree 24

JSON JavaScript Object Notation 6, 7, 19

LHC Large Hadron Collider 1

LSTM-AE Long-Short Term Memory Autoencoder 23, 40

OS Operating System 6, 7

RAM Random-Access Memory 2, 14

SWAN Service for Web based Analysis 9

TPR True Positive Rate 38

VM Virtual Machine 2, 14

WLCG Worldwide LHC Computing Grid 1, 2, 5

Appendix A

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
src ......................................... the directory of source codes
  implementation ............................ implementation sources
  thesis ............ the directory of LaTeX source codes of the thesis
text ............................................. the thesis text directory
  thesis.pdf ........................... the thesis text in PDF format
```