FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Fault detection in operations data
Student:	Ostap Iudin
Supervisor:	Ing. Jaroslav Kuchař, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

Monitoring operations data on servers and their intelligent application (AIOps) is an existing challenge. The goal of this thesis is to focus on collecting metrics about the state of devices, storing data as time series, using it for fault detection, and subsequent warning of possible problems.

- Get acquainted with the issues of infrastructure monitoring.
- Develop and test a tool to collect, store and pre-process data for a large number of simultaneously running machines. Focus on storing as time series, optimal data redundancy or a form of storage.
- Explore approaches to detect failures, especially from the point of view of detecting anomalies in time series.
- Choose suitable approaches, use existing implementations, or create your own implementation. Focus also on optimizing for a large number of controlled machines.
- Conduct experiments and evaluate the quality of the selected methods.
- Interpret the outputs appropriately to warn of possible malfunctions.

Electronically approved by Ing. Karel Klouda, Ph.D. on 10 February 2022 in Prague.

Bachelor's thesis

MONITORING SYSTEM AND DETECTING ANOMALIES IN SERVERS TRAFFIC DATA

Ostap Iudin

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Jaroslav Kuchař, Ph.D.
May 11, 2022

Citation of this thesis: Iudin Ostap. *Monitoring system and detecting anomalies in servers traffic data.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

List of Figures

List of code listings

I would like to thank the people who help me. Basically my supervisor Ing. Jaroslav Kuchař, Ph.D. Also my parents: Yuri Iudin and Svetlana Pashina. And my friends who were next to me during the difficult time of studying at the university: Daniel Poletaev, Aidar Mannanov, Vitaly Shakhmatov, Ivan Desiatov and Vladislav Krylev.

Declaration

FILL IN ACCORDING TO THE INSTRUCTIONS. VYPLŇTE V SOULADU S POKYNY. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue. Donec ipsum massa, ullamcorper in, auctor et, scelerisque sed, est. In sem justo, commodo ut, suscipit at, pharetra vitae, orci. Pellentesque pretium lectus id turpis.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue. Donec ipsum massa, ullamcorper in, auctor et, scelerisque sed, est. In sem justo, commodo ut, suscipit at, pharetra vitae, orci. Pellentesque pretium lectus id turpis.

In Prague on May 11, 2022 .....

Abstrakt

Tato bakalářská práce je zaměřena na vytvoření plnohodnotného nezávislého systému sledování serverů, vhodné pro použití v jakékoliv společnosti, která vlastní libovolným počtem zařízení, a využití strojového učení pro detekci anomálií v provozu technických zařízení k odstranění problémů v co nejkratším čase. V důsledku toho se mi podařilo vytvořit monitorovací systém, který dokáže sledovat stav více než 6000 zařízení a automatický systém pro detekci anomálií schopný předpovědět selhání práce 6 hodin před samotným selháním.

Klíčová slova monitoring, anomaly, detection, Grafana, Prometheus, alert, Promscale, pycaret, fbprophet, InfluxDB, Time series

Abstract

This bachelor's thesis is aimed at creating a full-fledged independent server monitoring system suitable for use in any company that owns any number of devices, and using machine learning to detect anomalies in the operation of technical devices to eliminate problems as soon as possible. As a result, I managed to create a monitoring system capable of monitoring the status of more than 6000 devices and an automatic anomaly detection system capable of predicting a failure of 6 hours before the failure itself.

Keywords monitoring, anomaly, detection, Grafana, Prometheus, alert, Promscale, pycaret, fbprophet, InfluxDB, Time series

Abbreviations list

CPU	Central processing unit
YAML	Yet Another Markup Language
SQL	Structured query language
PromQL	Prometheus Query Language

Chapter 1
Introduction

Nowadays, many large companies have a large number of devices that support their operation. Any malfunction in the operation of the devices can cause great harm and loss to the company. While the company is trying to find the source of the problem and fix it, it may lose a large number of customers. Therefore, I want to create a system that will minimize the probability of such an event. The purpose of this work is creating a full-fledged independent server monitoring system suitable for use in any company that owns any number of devices, and using machine learning to detect anomalies in the operation of technical devices to fix problems and eliminate them as soon as possible.

Objectives of the thesis: Flexible monitoring system. Automated fault detection system. Alert system that allows you to quickly solve the problem.

- In the Data collection section, I solve data collection problems, such as automatically identifying and finding the company's devices for data collection using the installed exporters on each of the devices and the *AutoDiscovery* script, which automatically creates a list of targets.
- In the Data storage I solve the problem of optimal storage of the received data about the operation of a large number of devices.
- In the Alerting section, I solve the problem of identifying problems using static defined rules. They also solve the notification problem by automatically creating a message in chats and creating an issue in Github with the definition of a person to solve the problem.
- In the Anomaly Detection section, I conduct an experiment in which I compare more than 10 algorithms for detecting anomalies, find the most suitable one and use it to detect anomalies in real time.

Chapter 2

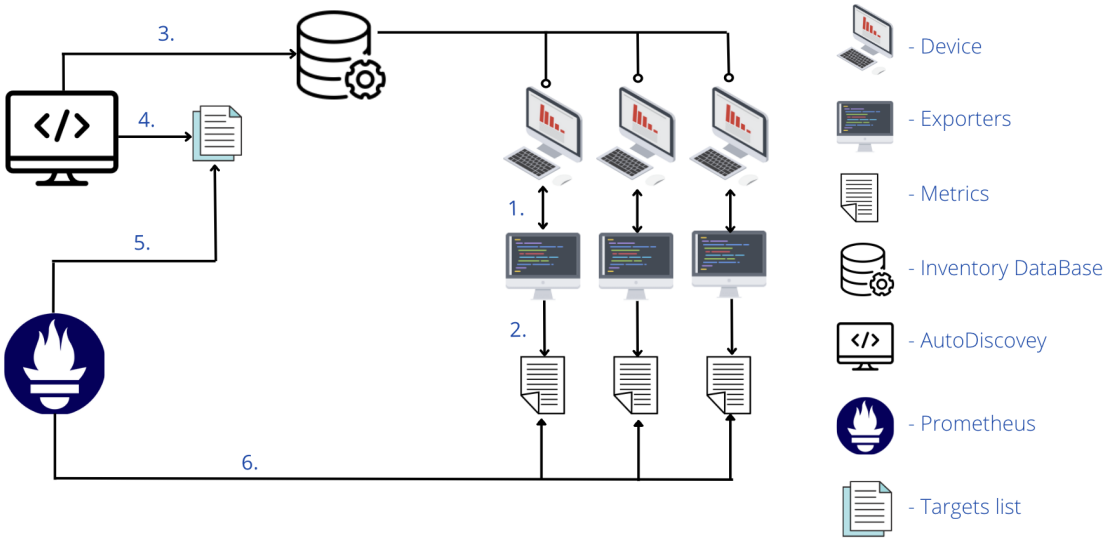
Collection of data

For monitoring, the first important task is to collect data, which is what I'm going to do in this section. I have created a flexible monitoring system that can be used in each company with configuration changes. The architecture provides methods for collecting data from each device and further managing the collected data.

In short , the collection scheme 2.1 consists of three components:

- Each of the devices has "Exporters" installed - programs that collect data about the device.
- *Prometheus* is a service that collects and combines data from all devices.
- *AutoDiscovery* is a program that automatically creates a list of all devices for monitoring.

Figure 2.1 Data collection scheme



2.1 Exporters

It was necessary to receive data about the operation of the devices. The ideal solution for this was the open-source *Node Exporter* program - a program for exporting data in the form of metrics - numerical measurements in the form of time series. Time series means that changes are recorded over time. For a web server, this may be the request time. For a database, this can be the number of active connections or the number of active requests [1]. But *Node Exporter* did not perform specific monitoring requests, such as getting information about the location of devices or the name of the team to whom this device belongs. So I wrote an additional functionality that exports local metrics.

2.1.1 Hardware Exporter

Hardware Exporter is a program written in Python for the special needs, collects local data that *Node Exporter* does not pay attention to but which are essential for the company, for example, which department this device belongs to, the serial number of the device, the location of the position in the data center. To control updates was added the export of the installed version of the *Hardware Exporter*. After collecting, the *Hardware Exporter* creates a file with these metrics, which will read *Node Exporter*. The path to the file must be specified in the *Node Exporter* configuration, *Node Exporter* must have the rights to read this file, and the metrics must be in the particular format drawn by *Node Exporter*em [1]:

```
metric_name{label_key1 = label_value1, label_key2 = label_value2, ...} metric_value
```

Listing 2.1 Metric format

2.1.2 Node Exporter

Node Exporter is an open-source program written in Go that collects various metrics, such as disk temperature, CPU load, network load... For each metric, it is possible to set scrape_interval - how often *Node Exporter* will update the metric value. It is also possible to configure metrics that *Node Exporter* will ignore. So was stopped following many unnecessary metrics that would reduce the load on the server and not take up memory. After collecting the metrics, *Node Exporter* combines them with the *Hardware Exporter* metrics.

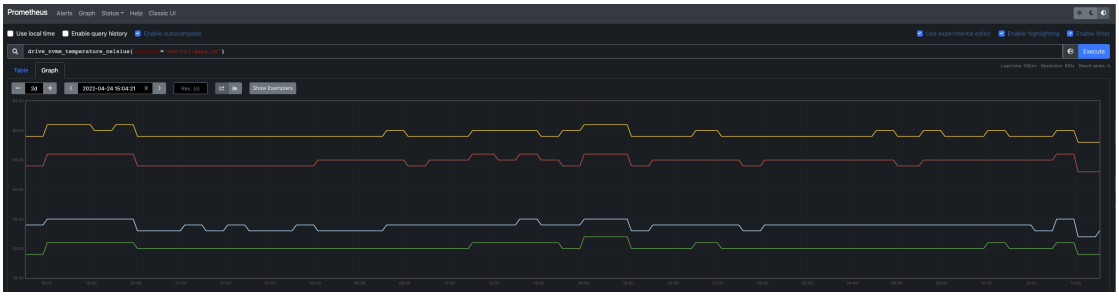
```
drive_temperature_celsius{SN="S465930",capacity="1920383410176",protocol="NVMe",type="nvme"} 30
drive_temperature_celsius{SN="S411937",capacity="1920383410176",protocol="NVMe",type="nvme"} 31
go_gc_duration_seconds{quantile="0"} 4.6989e-05
go_gc_duration_seconds{quantile="0.25"} 8.9057e-05
go_gc_duration_seconds{quantile="0.5"} 0.00012571
go_gc_duration_seconds{quantile="0.75"} 0.000165834
go_gc_duration_seconds{quantile="1"} 0.0002714
go_gc_duration_seconds_sum 122.994432347
go_gc_duration_seconds_count 942139
go_goroutines 11
```

Listing 2.2 Metric examples

2.2 Prometheus

To collect device performance data, I chose *Prometheus* because it has excellent documentation and a large community ready to help with any problem that arises. *Prometheus* is an open-source monitoring system. *Prometheus* collects and stores metrics in the form of time series data - information about metrics is stored with the timestamp in which it was recorded, along with optional key-value pairs called labels[2]. Also, with the help of *Prometheus*, it is possible to build simple graphs using a PromQl request.

Figure 2.2 Prometheus graph



The principle of operation of *Prometheus* is that it sends get requests to devices that are defined in the configuration as targets. The targets can be written directly in the configuration itself or specify the path to the file with the targets or the URL where is the list of targets.

```
scrape_configs:
 - job_name: 'node' # the name will be added to the metrics label
  scrape_timeout: 30s # how often to send a request to targets to get metrics
  file_sd_configs: # files with list of targets
  - files:
   - 'target.json'
   - 'targets.d/localhost.json'
```

Listing 2.3 Prometheus configuration

Prometheus is also a database with a query language - PromQL (Prometheus Query Language) that lets the user select and aggregate time-series data in real-time.

```
sum(http_requests_total{method=\GET"} # Returns the number of all http get requests
```

Listing 2.4 PromQL example

2.3 Autodiscovery

Each company with a more significant number of devices keeps an inventory. All devices will be stored in a database and may be accessible via a web service. To combine this data and monitoring, was written the program - *Autodiscovery*. The task of *Autodiscovery* is to configure

targets and update them automatically. It is possible to configure the search parameters in the configuration file in YAML format. The program parameters can be configured by filling arguments in the command line to run the program.

For greater flexibility, the program was written so that to change the final list of targets, it is enough to change the configuration file and there is no need to change the code. In configuration *Autodiscovery*, it is needed to fill in the URL to the database of data or the Web service API, fill in the token or login/password to the service from where the monitoring devices will be taken. *Autodiscovery* allows you to configure filters for selects by which the target list will be created. It is possible to specify regex for hostname, shortname, serial number, team, and position. To use the ”or” operator in select, you need to set the number of select in exports_number and fill in the required number of filters, and all filters will be combined using the ”or” operator. If all conditions are set in one filter, they will be combined by the ”and” operator. To further simplify the target search process, a new ”monitoring” label has been added to the device database. *AutoDiscovery* is able to filter targets by location and given label. *Autodiscovery* startup arguments require a location and an ordered number of target parts.

Horizontally scaling

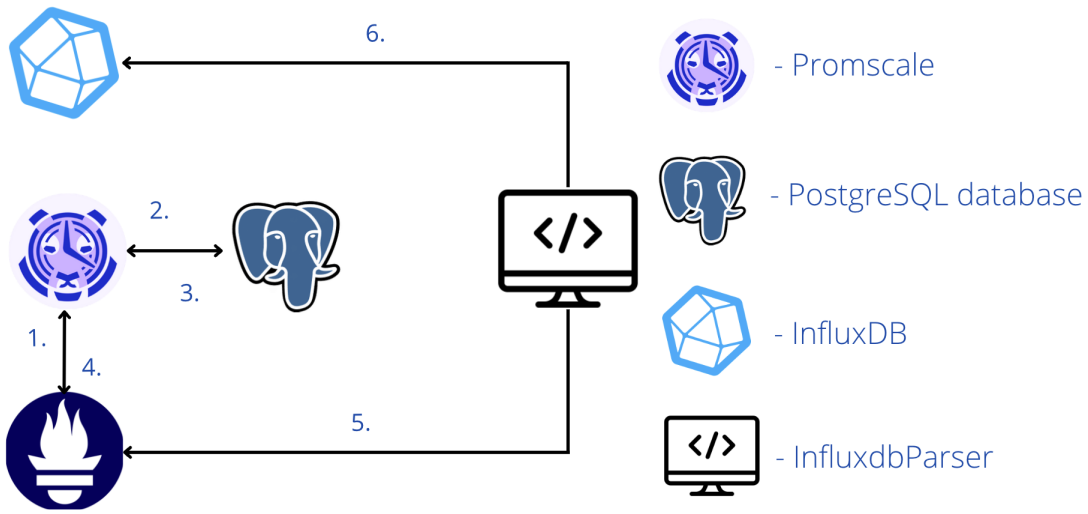
To optimize the work of monitoring with a large number of devices, more specifically, to optimize the load on Prometheus by reducing the number of targets the ability to configure the separation of targets has been added. If there are many monitoring objects in one location, it is possible to scale horizontally by adding another monitoring host and changing the ”separate_to” parameter in the config. You will also need to specify the monitoring host sequence number in the *Autodiscover* startup argument. After that, each target list will be divided into an equal number of targets for monitoring and each host monitors a given part of the targets.

Chapter 3

Storage of data

The data obtained during the collection process must be stored. I use two databases for this 3.1. *Promsclae*, which is ideally combined with *Prometheus*, since in addition to SQL it supports PromQL. And *InfluxDB* for storing string data, since *Prometheus* can only work with numeric values.

Figure 3.1 Data storage scheme

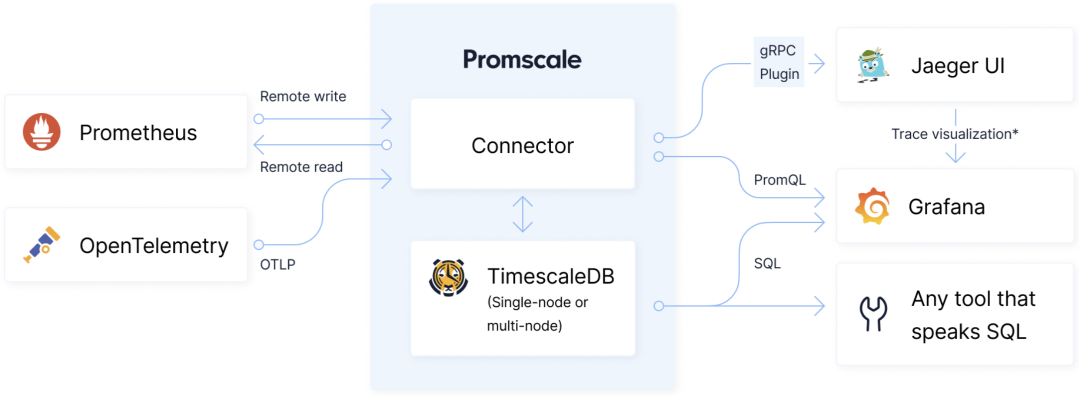


3.1 Promscale

The next part of monitoring that is important for analysis and intelligent use is the database mentioned above - *Promscale* 3.2. [3]

Promscale

Figure 3.2 Promscale scheme



Promscale is an open source observability backend for metrics and traces powered by SQL. Collector and is 100% PromQL compliant.

The use of *Promscale* as a long-term database instead of alternatives is due to 2 things. The first is that the *Promscale* community is quite large, and there is always a place where it is possible to ask or consult in case of any problem. The second is the ability to use PromQL for queries, which greatly facilitated data acquisition for analysis and anomaly detection. For security reasons, when working with the *Promscale* API or using *Promscale* as a data source from *Grafan* in a remote location, the TLS protocol is used, and *Promscale* is only accessible via HTTPS.

3.2 InfluxDB

The last part of monitoring that ensures the entire working of monitoring is *InfluxDB* [**InfluxDB**].

Using *InfluxDB* alone is not optimal because, firstly, it does not support SQL or PromQL, and secondly, it is not free. The features of the free version are quite enough to work only with string data, so it was chosen for these purposes.

InfluxDB is a time series database designed to handle high write and query loads. It is an integral component of the TICK stack. *InfluxDB* is meant to be used as a backing store for any use case involving large amounts of timestamped data, including DevOps monitoring, application metrics, IoT sensor data, and real-time analytics.

The reason for using the second long-term database is that *Prometheus* cannot work with text data as with the value of metrics. However, text data, such as the department that the device belongs to, the bios version, or the firmware version, are fundamental. Therefore, it is necessary to add a database that could work with this. A separate program that works on monitoring host pars labels from *Prometheus* and sends them to *InfluxDB*. *InfluxDB*, in turn, is a data source for *Grafana* [4], thanks to which it is possible to control many text parameters.

3.3 InfluxDB parser

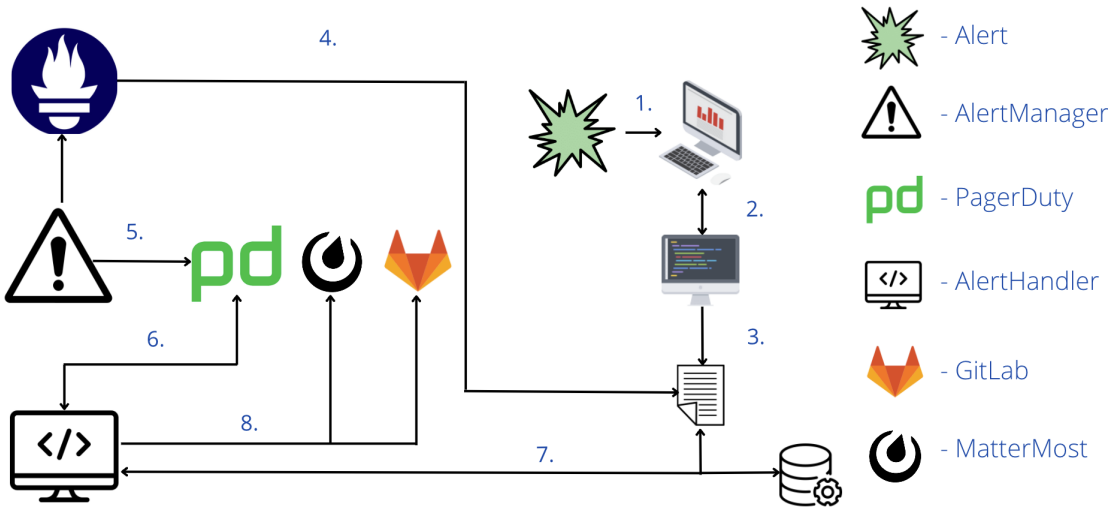
The next small but essential program for monitoring is the *InfluxDB* parser. Its purpose is to get string data from labels metrics and write them to a database that supports *InfluxDB* string data.

Chapter 4

Alerting

To prevent serious problems, I monitor some important parameters using static rules. For example, I look to ensure that the battery voltage is not below 2.6 volts. For tracking and primary processing of these values, I use Alert Manager. After the alert occurs, it is processed a second time by the *AlertHandler* program implemented by me, which adds information to facilitate the identification of the problem and informs about it in the chat and creates an issue in Gitlab 4.1.

Figure 4.1 Alert scheme



4.1 Alert manager

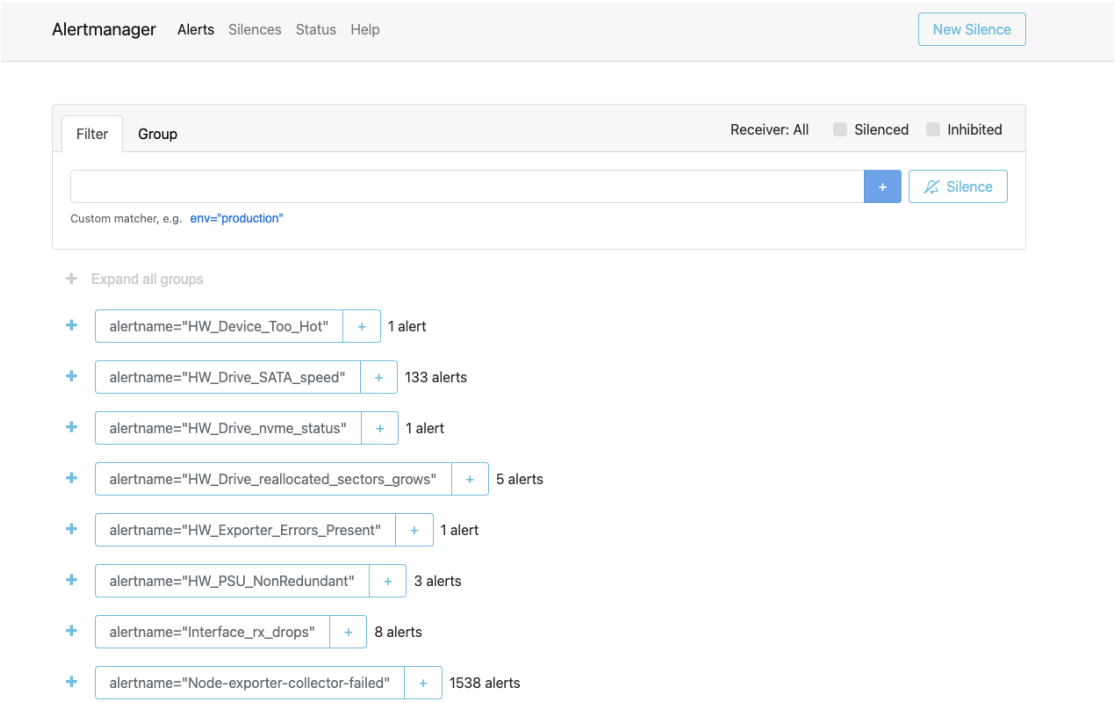
The next monitoring service is Alertmanager [5]. The Alert manager handles alerts sent by client applications such as the *Prometheus* server. It takes care of deduplicating, grouping, and routing them to the correct receiver integration such as email, PagerDuty [6], or OpsGenie [7]. It also takes care of silencing and inhibition of alerts. The URL to Alertmanager is specified in the *Prometheus* configuration. When alerts occur, *Prometheus* sends them to the Alertmanager.

That eliminates duplicates and processes and groups them according to the rules written in the configuration 4.2.

```
route:
  group_by: [alertname]
  receiver: 'pagerduty-notifications'
  group_interval: 5m
```

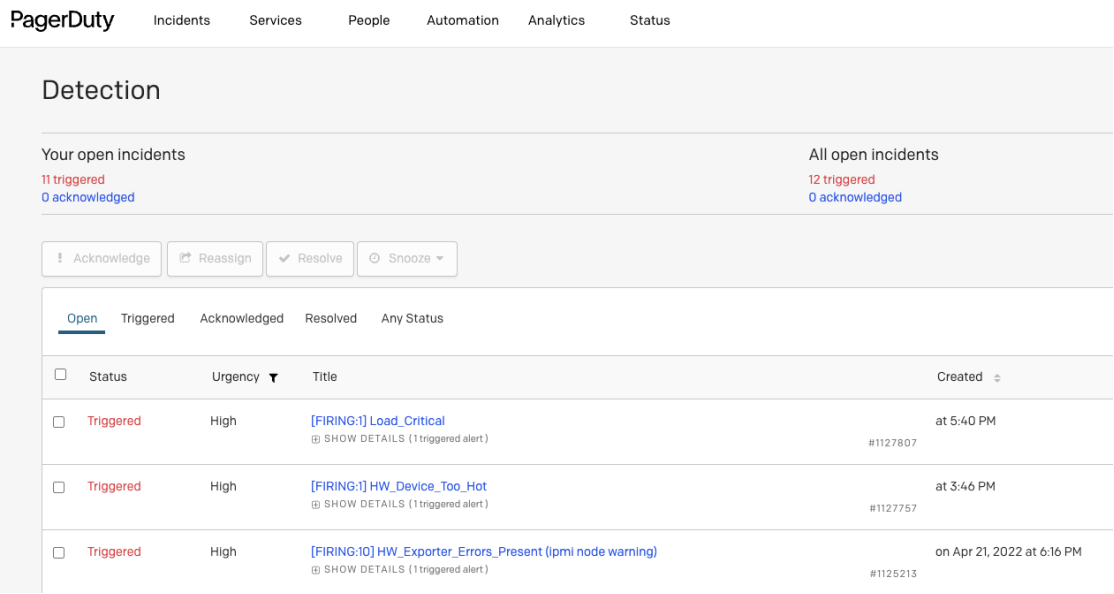
Listing 4.5 AlertManager configuration

Figure 4.2 Alert manager



Alert manager allows integration with different services. In monitoring, it was decided to use integration with PagerDuty. I chose PagerDuty Because it has a convenient API and web interface. PagerDuty is an online incident handling platform. When a new alert appears, Alertmanager sends it to PagerDuty, which creates an incident and, depending on the escalation policy, sends an alert to the mail or a messagecall to the person's mobile phone is responsible during the incident. PagerDuty provides many different options for setting the frequency and method of warning 4.3.

Figure 4.3 PagerDuty



4.2 AlertHandler

An equally important part of monitoring is alerting. To simplify the understanding of alerts and to process them more conveniently, the *AlertHandler* program was written. This program performs several essential functions at once 4.4.

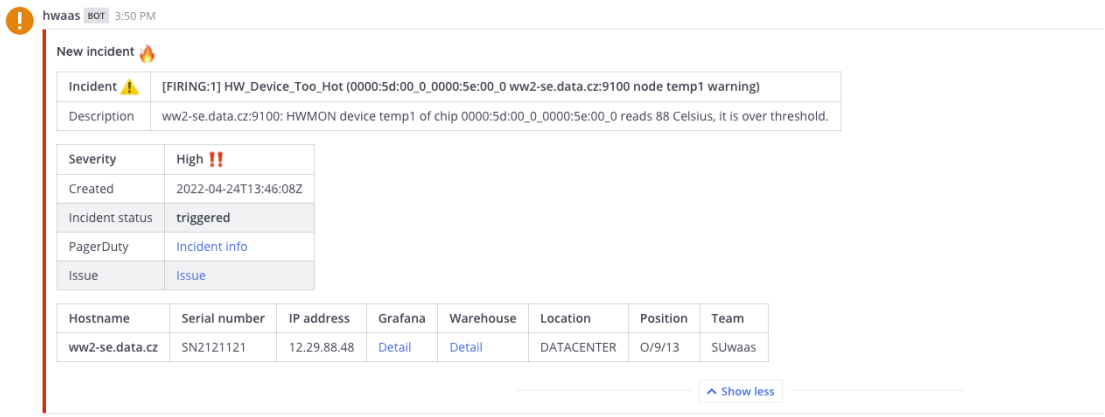
Figure 4.4 Alert scheme

Adding the necessary information

The first task of the alert handler is to track the appearance of new alerts. *AlertHandler* checks the list of active incidents in PagerDuty. When a new incident occurs, the program refers to the exporter of the node of the device on which the incident occurred. There *AlertHandler* will take the serial number information to identify the device and the team to which this device belongs. Then the program will request information about it from the database where all the

company's devices are stored. In addition, *AlertHandler* uses the information received to create links to *Grafana* on the Detailed Control Panel of this device. As a result, *AlertHandler* adds the necessary information to speed up the problem solving process, such as serial number, IP address, link to *Grafana*, link to web inventory, location, position in the data center, team. With all this information, the program creates a convenient table in markdown format by contacting a special channel and creates an alert there about the occurrence of a new incident.

Figure 4.5 New Incident in Mattermost



Incidents history

The second thing that *AlertHandler* allows us to do is the ability to track the history of incidents. The program reports a new incident immediately. If the incident has not been resolved within a specific time, *AlertHandler* will send a warning that the problem has not yet been resolved 4.6. The time between repeated warnings depends on the level of criticality of the incident. Also, when the incident is resolved, the *AlertHandler*ler will notice it and send a laudatory notification about it 4.7.

Figure 4.6 Incident was not resolved

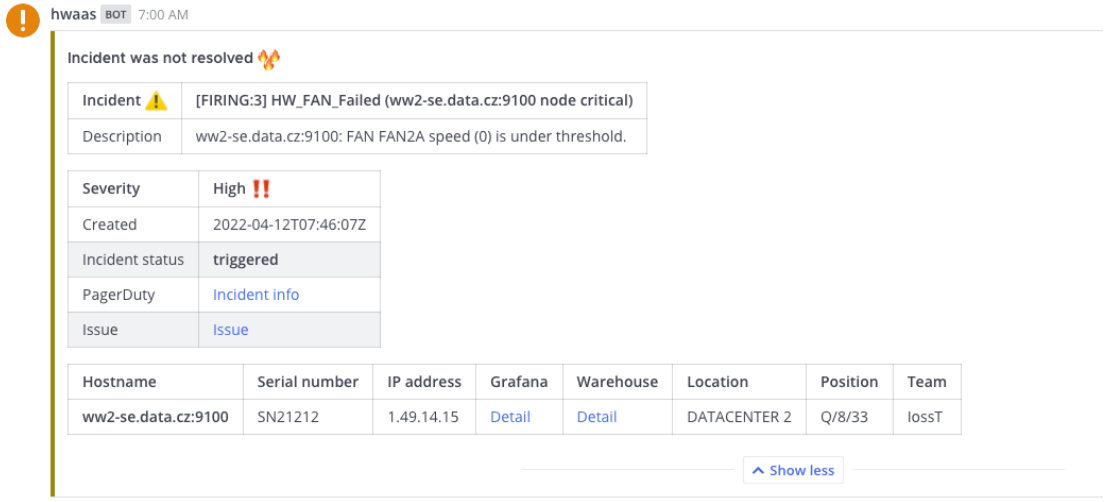
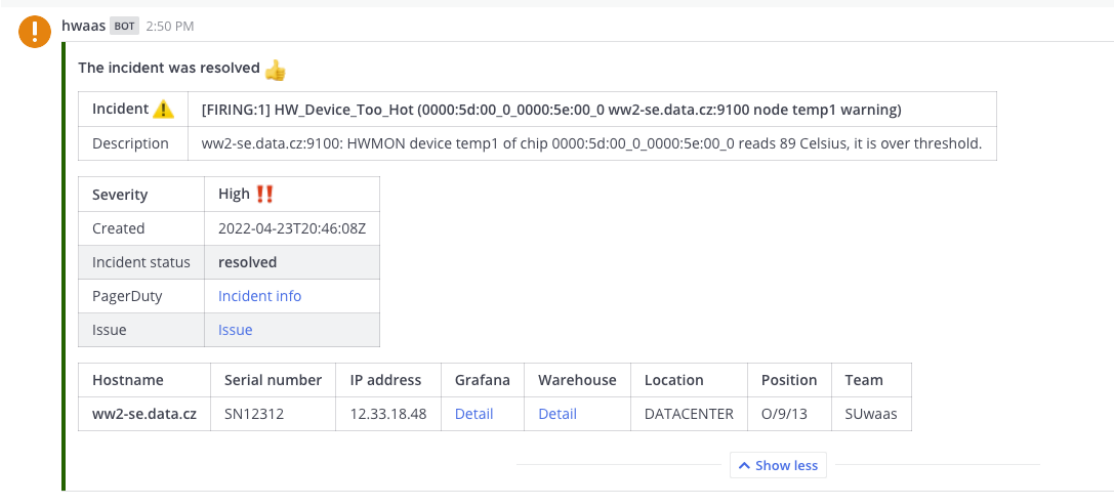


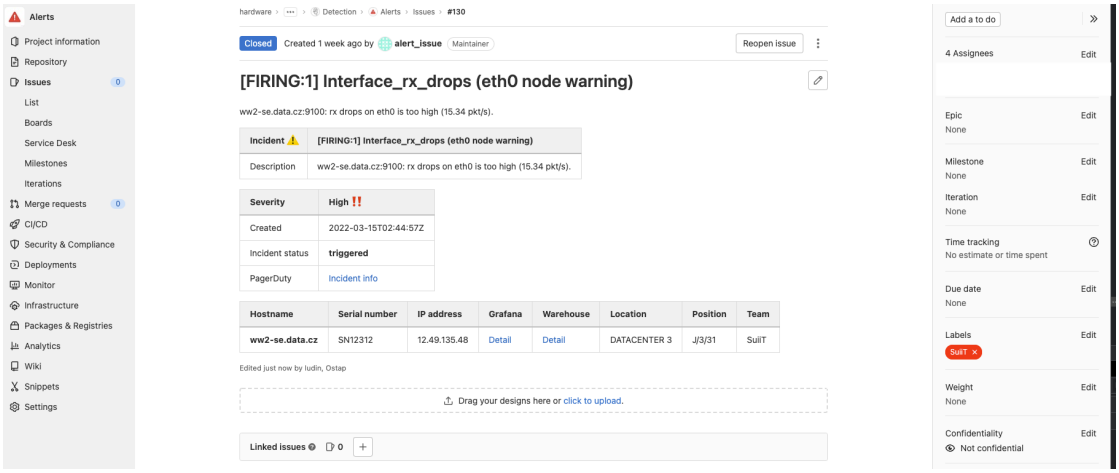
Figure 4.7 Incident was resolved



GitLab issue

The next thing that simplifies the solution of the problems that have arisen is the automatic generation of the issue in GitLab. After receiving additional information (the process was described in the previous part), *AlertHandler* creates an issue in a specially created project. The issue description contains all of the above parameters of the problematic device extracted by AlertHundler. The issue 4.8 is always assigned to a person from the supervision team and to a person who is a representative of the team that owns the device. Also, to find the issue faster, each issue will be labeled with a label with the team's name. This will do possible to quickly get a list of team's issues using label filtering.

Figure 4.8 Alert issue



4.3 Visualization

To monitoring the operation of the device, some kind of visualization is needed. One of the best open-source projects for data visualization, including time-series data, is *Grafana*. When an

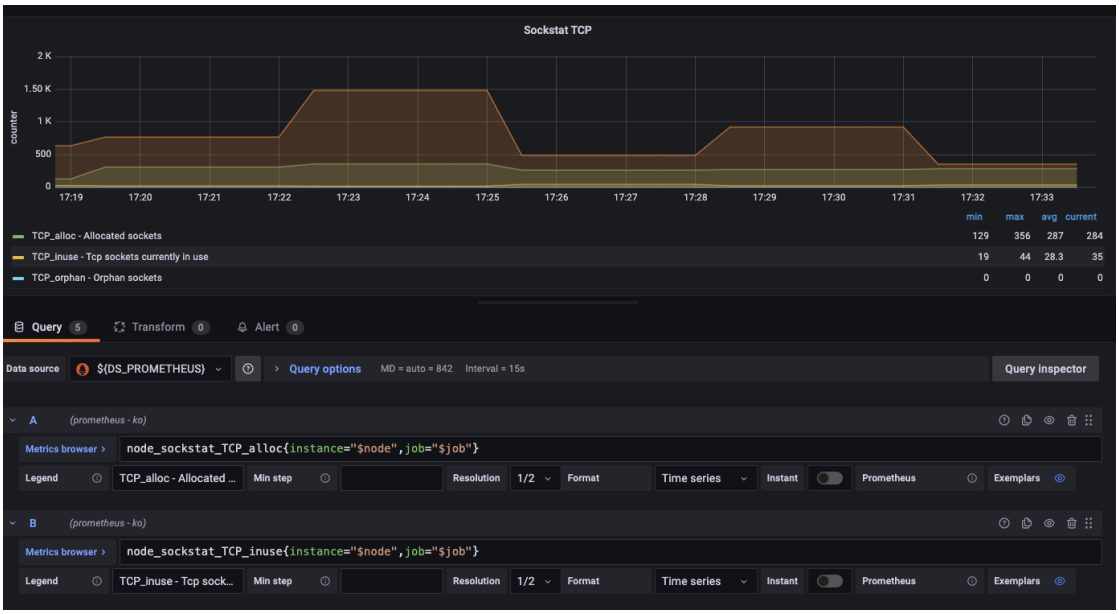
alert occurs, the alert will contain a link to the graphs of the device in which the failure occurred 4.9. Thanks to which it will be possible to track the trend of changes in this metric and better understand the situation to determine the cause of the failure.

Figure 4.9 Grafana dashboard

Grafana

In monitoring, *Grafana* is used to display metrics data in tables, graphs, and other suitable forms [4]. In *Grafana*, it is possible to create dashboards that will contain panels with graphs and tables 4.10. To create a panel is needed to select the visualization form, data source, and write select. If the data source is *Promascale* or *Prometheus*, select should be in PromQL.

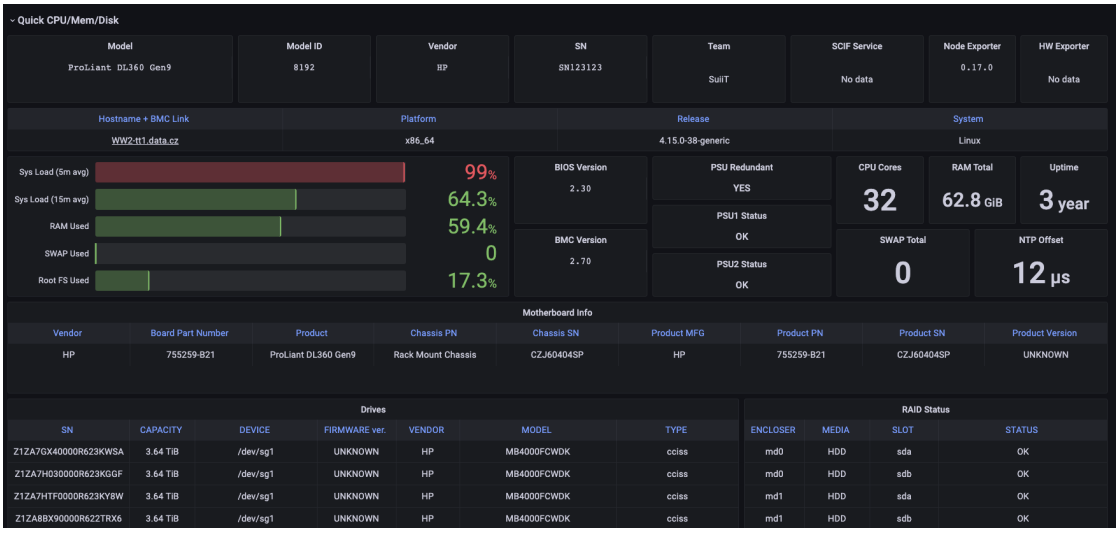
Figure 4.10 Grafana dashboard



Below are some essential dashboards from monitoring:

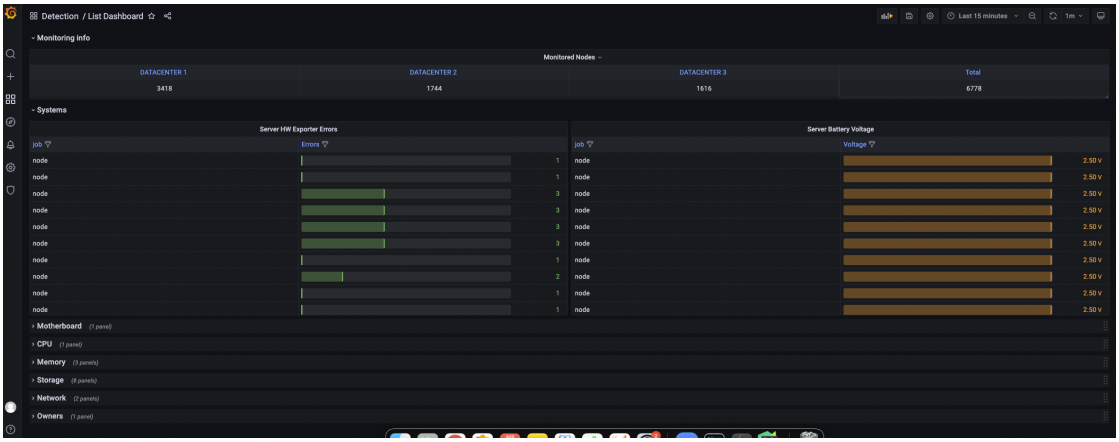
- **Detailed Dashboard** that dashboard is a summary of data in the form of tables and a bar gauge about a specific device.

Figure 4.11 Detailed dashboard



- **List Dashboard** is the second important dashboard. It contains tables about all devices in monitoring and their number.

Figure 4.12 List dashboard



Chapter 5

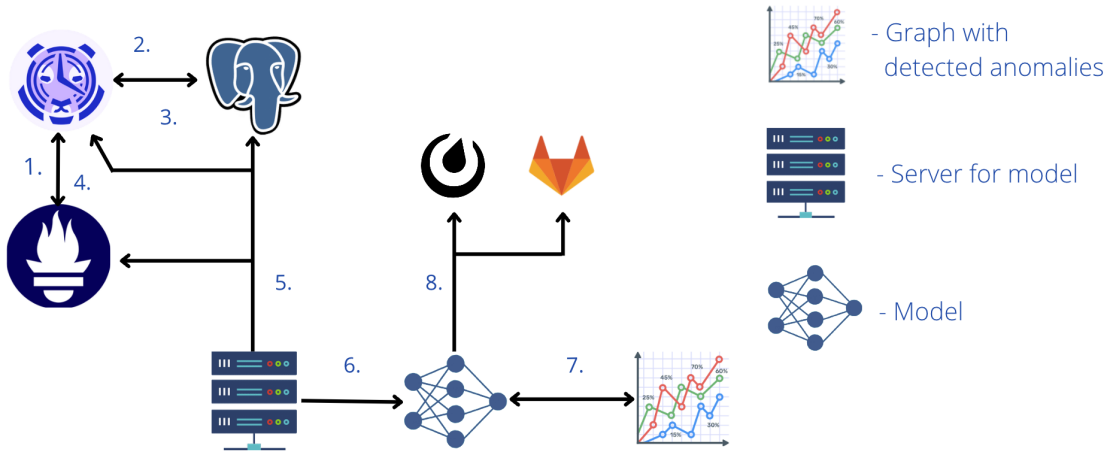
Anomaly detection

Anomaly detection is understood as the identification of rare elements, events, or observations that differ significantly from most data and do not correspond to a well-defined concept of normal behavior.

The most significant potential of the entire monitoring system is using machine learning to analyze the collected data. I have about 20 alert rules written, which is too little to use all the monitoring features. Considering that the number of metrics exceeds 150, 20 alert rules do not use even 20%. It should also be borne in mind that the alert rules were written by people based on their experience and expertise and cannot consider all possible scenarios of the device and will not be able to report a serious problem. Therefore, the use of artificial intelligence as a more objective and flexible form of error detection is necessary.

To detect anomalies in the operation of devices, a script was written - Anomaly Detection 5.1.

Figure 5.1 Anomaly detection scheme



The script takes data from a source specified in the configuration, learns from this data or

detects anomalies using a scientific model, stores a graph showing normal behavior and behavior taken for an anomaly and reports in case of detection.

The program is written in Python and has its configuration file in which it is possible to specify all the essential parameters for detecting anomalies.

5.1 Data source

The script should first get data to train the model or use a ready-made model to detect anomalies in this data. Anomaly Detection can use three data sources: *Prometheus*, *Promscale*, and *PromsclaeAPI*. Each of the data sources has its class written. The configuration needs to specify which data source will be used and set the necessary parameters for connecting to the selected source and parameters for receiving data.

After that, the selected class will be passed to the main function, and the data will be received depending on the specified parameters.

Each of the classes has a ”connect” method that connects to the database or simply checks the availability of the data source. Also, each of them has a ”get_data” function that receives data depending on the configured parameters in the configuration and returns them in the ”pandas.DataFrame” table format with time series data 5.1.

```
def main(
        database = None,
        dbname: str = None
):
    """
    Getting data and call detection function
    :param database: (class) Class for working with datasource
    :param dbname: (str) Database name
    """
    # Getting data
    if not CONFIG["only_read"]:
        database = database(**CONFIG[dbname]["connect"], logger=logger)
        if not database.connect():
            return False
        data = database.get_data(metrics=CONFIG["metrics"],params=CONFIG
    [dbname]["params"])
    else:
        data = read_data(dbname=dbname)
    # Detection
    for metric, table in data.items():
        detection(data[metric], metric, CONFIG["metrics"][metric])
```

Code listing 5.1 Detection code

The main parameters for obtaining data are set in the configuration. The whole process of data analysis is very flexible and automated. In order to add analysis to new data, in this case to a new metric, it is enough to set three parameters to the config, while the code does not need any changes.

Data source parameters

You need to define the metric by writing its name. Then it needs to write three parameters 5.6. ”Query”: PromQl query to get data in the desired format. ”Separator”: the parameter for dividing the metric into several by some label, can be left empty. This parameter was necessary because many metrics contain information about different components of the device.

For example, the ”node_cpu_second_total” metric measures CPU load, but it measures this for different mods. For example, ”user” or ”system” both mods have different trends and normal, non-abnormal behavior for the time. Therefore, it is better to divide this metric into several depending on the mode and analyze them independently of each other.

The third parameter is the value of the static alert. It is needed for a more informative graph. It will be shown on the graph along with the analysis results after detection. It is also a source of truth for determining anomaly detection quality for artificial intelligence models (more details later). The parameter can also be left blank.

```
metrics:
  # metric name
  node_cpu_seconds_total:
    # query for metric getting
    query: "sum by (mode)(rate(node_cpu_seconds_total
                    {instance=\"to_replace_instance\"}[8m])"
    # separator label name
    separator: mode
    # value of static alert
    static_alert: 420
  node_hwmon_temp_celsius:
    query: "node_hwmon_temp_celsius{instance=\"to_replace_instance\"}"
    separator: chip
    static_alert: 85
    ...
```

Listing 5.6 Metrics configuration

In the configuration it is possible to set the value of the ”read_only” parameter to ”True” and the path to the file with the previously received data in CSV format 5.7. It is possible not to specify or configure the data source in this case.

```
only_read: True
data_path: "data/node_cpu_seconds_total.csv" # path to data file
```

Listing 5.7 Data getting configuration

5.2 Prometheus - data source

Prometheus is the class implementing data source *Prometheus*. To connect to it, it needs to specify only the access URL. It is possible to specify the file to process the data to save it. If the ”save” parameter is set to false in the config, you can not specify the path to the file to save 5.8.

```
datasource: "Prometheus"
save: False
Prometheus:
  connect:
    url: "http://12.224.2.113:9090"
    save_to: "data/Prometheus/"
    params:
        label_config: {}
```

Listing 5.8 Prometheus - data source configuration

Prometheus class uses the package "Prometheus_api_client" to communicate with the data source. This package provides a function for translating exported data into the "pandas.DataFrame" format, which is what most anomaly detection models work with. Therefore, it is the most convenient DataSource for analysis and requires no additional modifications 5.2.

```
  metric_data = self.connection.custom_query_range(
            query=query,
            start_time=start_time,
            end_time=end_time,
            step=step # 240s
        )
  metric_table = MetricRangeDataFrame(metric_data)
```

Code listing 5.2 Prometheus getting data

5.3 Promscale - data source

The following class uses *Promscale* as a data source, more precisely, a PostgreSQL [8] database that uses by *Promscale*. The parameters needed to connect and work with this data source are also specified in the configuration. This is the host and port on which the PostgreSQL database is available. The connection parameters are also username, password, and database name. If the "save" parameter is set to True, you also need to set the "save_to" parameter with the path where the data will be saved 5.9.

```
datasource: "promscale"
save: False
promscale:
  connect:
    host: "12.239.21.42"
    port: 5432
    user: "postgres_user"
    dbname: "postgres_dbname"
    password: "db_password"
    save_to: "data/promscale/"
```

Listing 5.9 Promscale - data source configuration

The Promscale class uses the "psycopg2" library to work with a PostgreSQL database 5.3.

```
    cursor = self.connection.cursor()  # cursor for communicating with
    db
    self.logger.info(f'{metric} :', to_term=False)
    query = Query.get_metric(metric) # <- PostgreSQL query
    try:
        cursor.execute(query)
    except Exception as e:
        self.logger.info(f'Error. {e}', to_term=False)
        cursor.close()
```

Code listing 5.3 Promscale getting data

This data source is less convenient for analysis since additional processing of the received data is required. The resulting table contains labels as one common column, and it needs to be parsed into separate columns for each individual label. In addition, PostgreSQL does not support PromQL, which is why a class was created that returns the desired SQL query for each metric.

```
    table = Promscale.cursor_to_dataframe(cursor)
    table = self.label_to_columns(metric, table)
```

Code listing 5.4 Transform data to DataFrame

5.4 PromscaleApi - data source

The third class for getting data is PromscaleApi. Like the previous one, it also uses *Promscale*, but in this case, the program does not connect to the PostgreSQL database via "psycopg2". It uses the *Promscale* API to work with data source.

For correct connection, configuration needs to set precisely the same parameters as in the case when the data source is *Prometheus*. The URL, the path to the file to save, and additional parameters for the data specification to receive 5.10.

```
datasource: "promscaleapi"
save: False
promscaleapi:
  connect:
    url: "http://14.234.1.123:9090"
    save_to: "data/promscaleapi/"
    params:
        label_config: {}
```

Listing 5.10 PromscaleApi - data source configuration

The advantage of this approach before using a ready-made library to connect to PostgreSQL is two key things at once. Firstly, it is an opportunity to use PromQL for a query. PromQL is created specifically for convenient data retrieval from time series tables. It is a much more convenient tool in our case. The second point is the data format in response, as in the case when the data source is *Prometheus*, the data obtained can be transferred to pandas.DataFrame without problems and additional modifications are used for training and anomaly detection 5.4.

```
    url = self.url
    headers = {
        'Content-Type': 'application/json',
    } # request header
    params = {
        "query": query, # <- PromQl query
        "start": time.mktime(start.timetuple()),
        "end": time.mktime(end.timetuple()),
        "step": step
    }
    response = requests.get(  # request to PromscaleApi
        url=url,
        params=params,
        headers=headers,
    )
    result = response.json()['data']['result']
    table = pd.DataFrame(result, columns=['timestamp',  'value',
                                                    separator]) #
convert to Pandas table
```

Code listing 5.5 PromscaleApi getting data

5.5 Detection

After collecting the data, the program makes an detection. The detection process can be configured using the parameters in the configuration file. There are three parameters that can be configured in the configuration file 5.11.

The model can be overfitted, so it will not always be rational to use the old model and it will be necessary to learn the new one. So I added the following configuration parameter - "get_model". Depending on which the script will learn from the data anew or take the previously saved fully finished model or the parameters of the finished model from the file and use them to detect anomalies.

The second parameter is "save_model", which is responsible if the program saves the model for detecting anomalies and writes it to the file.

The third parameter is "model_path", which should contain the path to the file where the model will be written or from where the program will take the finished model for detecting anomalies.

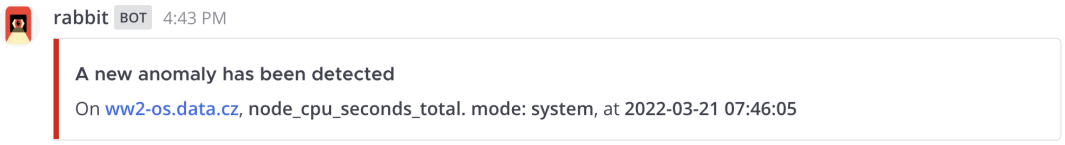
```
    save_model: True
    get_model: True
    model_path: "Detection/model"
```

Listing 5.11 Model configuration

After preparing the data, the program calls the "detection" function. The selected model processes the data, and at the output, we get a new table with the indicated anomalies and a graph in which all this is clearly depicted.

The model receives new data, learns, does detection, and improves every time. If anomalies are found, the program will report the time and metric in which the anomaly was found through a special channel. 5.5.

Figure 5.2 Anomaly alert



Optimization

When choosing the algorithm, the speed of learning and detection was also taken into account. The chosen algorithm works fast enough for 6000 devices. Learning and detection for one device according to data from a week ago will last only 0.4 seconds. Detection on all 6000 devices takes about 40 minutes.

Chapter 6

Experiment

I tried to use two frameworks: *fbprophet* [9] and *pycaret* [10]. A total of 11 different models in order to find the most suitable for detecting anomalies in monitoring.

6.1 Model scoring

The learning of the anomaly detection model on the data obtained from monitoring is unsupervised learning because we are not given an exact explanation of what an anomaly is and normal behavior. Therefore, the judgment on the quality of detection could only be judged on my expert opinion. However, I wanted to express this judgment in numbers. To do this, additional functionality was written that returned an estimate of the final detection for each model. The source of truth for this is a time-tested and compiled of many years of experience - static alert. Based on this, the judgment about the quality of detection is based.

The function accepts a data table after detection, that is, in which there is an ”anomaly” column that indicates whether the model considered this value abnormal or normal. The second parameter is the value of a static alert, at the intersection of which an incident will be created, and the monitoring team will be alerted about it. And the third parameter is if the behavior under a static alert is considered normal or above it.

To simplify, I will demonstrate everything with an example of when the behavior under the static alert line is considered normal. Everything is likewise in the case when the behavior is considered normal if the value does not fall below the static alert line.

Except two models (about them later), all recognized abnormal behavior in values exceeding the static alert line. Therefore, the ”calculate_score” function considers the quality of anomaly detection under the static alert line for a more objective assessment. In an ideal situation, the model would have to warn that the line might be crossed in the near future, and the monitoring team would have to react somehow to prevent this. At the same time, the model would not have to react to values in the ”trust” interval, thereby not creating panic from scratch. Therefore, the ideal method for evaluating models is to compare the distance between the line of static alert and the line of the average value of all found anomalies. Thus, we reward the model for anomalies near the static alert line and penalize it for the value of normal behavior as an anomaly 6.1.

```
def score_for_detected(
        anomaly: pd.DataFrame = None,
        static_alert: float = 0.0,
        under: bool = True
):
    """
    :param anomaly: (pd.DataFame) Table of all detected anomaly
    :param static_alert: (float) Value of static alert
    :param under: (bool) If point under alert is it normal
    :return: (float) Score for detected points
    """
    anomaly_over_alert_line = anomaly.loc[anomaly.value >
    static_alert].value  # values of anomaly points over line
    anomaly_under_alert_line = anomaly.loc[anomaly.value <
    static_alert].value. # values of anomaly points under line

    if under:
        #  if static warnings check a value above a certain value, it
     means that score counts for detection under the static warning
    line
        anomaly_mean = anomaly_under_alert_line.mean()
    else:
        anomaly_mean = anomaly_over_alert_line.mean()

    anomaly_point = abs(static_alert - anomaly_mean)
    return 100/anomaly_point # <- value normalization
```

Code listing 6.1 Score calculate

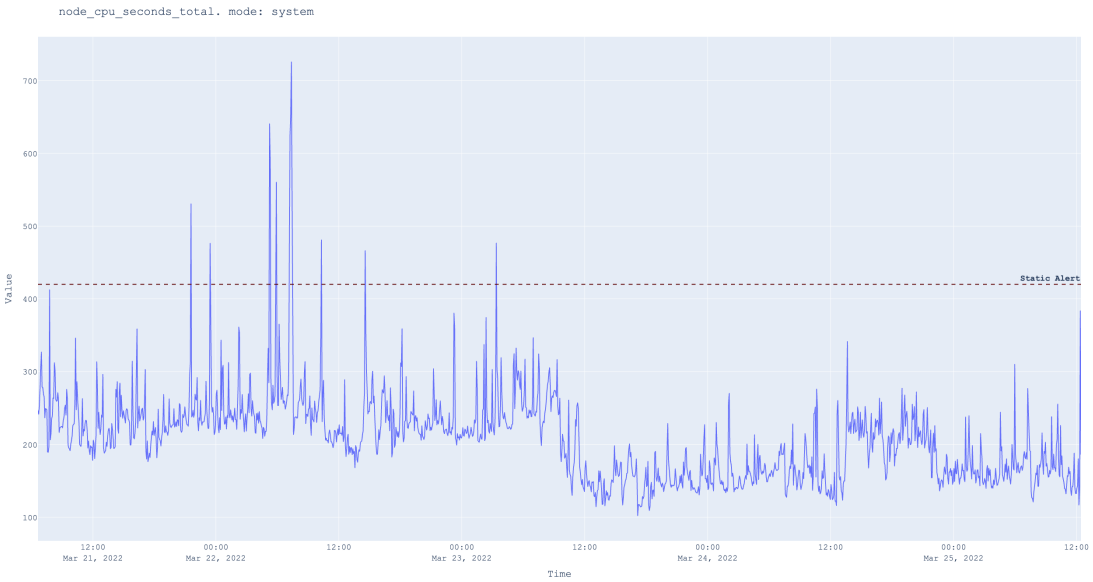
6.2 Model comparison

For comparison, I took the data for the last week from the ”node_cpu_seconds_total” metric in the system mode, which shows the CPU load. The principle is absolutely the same about other metrics and different instances. A static alert rule for this metric restricts normal behavior from above to a value of 420. For this metric, I trained all models, drew a graph with the results, where normal behavior will be shown by blue lines and abnormal activity by red dots, and calculated each score. Below are two graphs on which you can clearly compare all tire models by their score, and the time of training and detection 6.1.

Node cpu seconds total

First, let us look at the metric itself and its behavior during the week. Also, on the graph, you can see the static alert line and the values that intersect it.

Figure 6.1 Node cpu seconds total

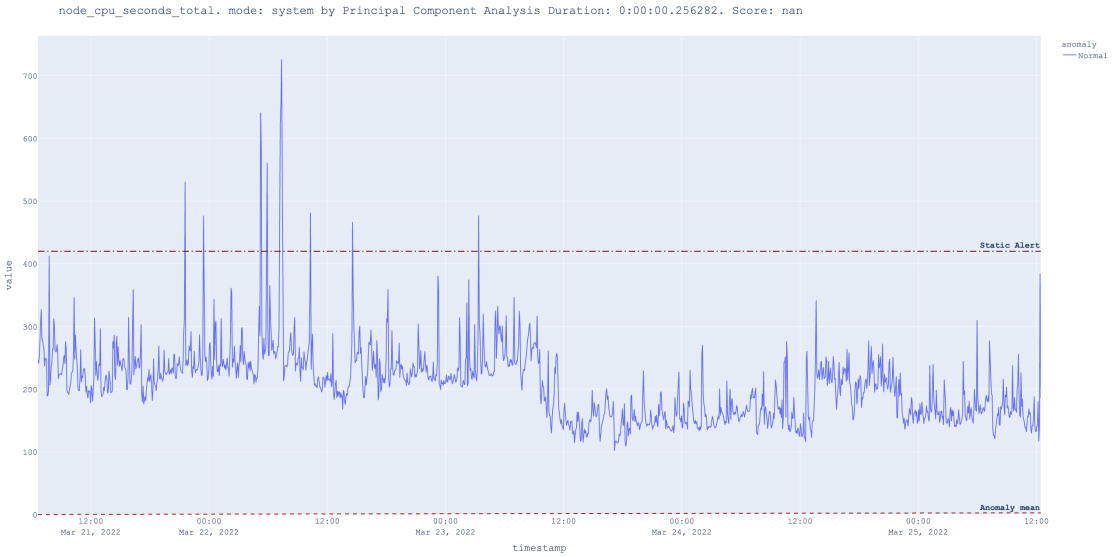


Now let us take a look and analyze each of the models and their results in more detail. Let us start with the more unsuitable ones to the better ones.

6.2.1 Principal Component Analysis

The idea is to split the data into its main components and then restore the original data using only the first few main components [11]. The recovered data will not be the same as the original data. The recovered data elements that differ most from the corresponding source elements are anomalous. Applying this approach to monitoring data is not good. Because this model, like the next two, did not find abnormal behavior in the data at all 6.2.

Figure 6.2 Principal Component Analysis



6.2.2 Subspace Outlier Detection

Stochastic Outlier Selection - his uncontrolled outlier selection algorithm takes either a feature matrix or a difference matrix as input and outputs the probability of outliers for each data point [12]. Intuitively, a data point is considered an outlier when other data points have low similarity.

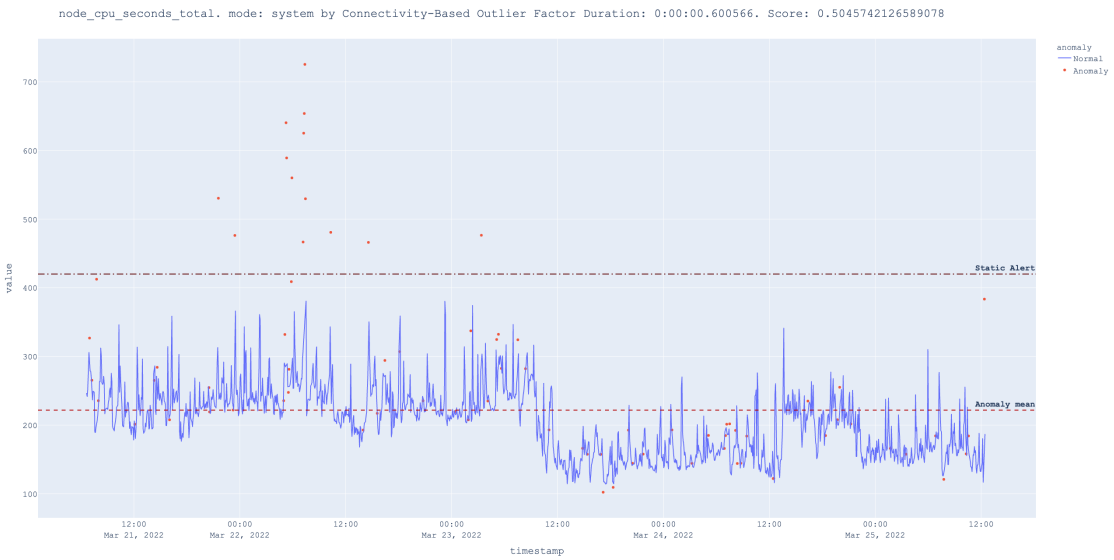
The algorithm also did not detect abnormal behavior, so it is not applicable in this case.

6.2.3 Connectivity-Based Outlier Factor

The first algorithm that begins to identify anomalies in the data is the Connectivity-Based Outlier Factor [13].

Connectivity-Based Outlier Factor - it is a method for detecting outliers. The idea behind the join-based radiance algorithm is to assign a radiance to each data point. The third emission power is the compound-based emission factor, COF data points. A high COF value of a data point represents a high probability that it is an outlier 6.3.

Figure 6.3 Connectivity-Based Outlier Factor

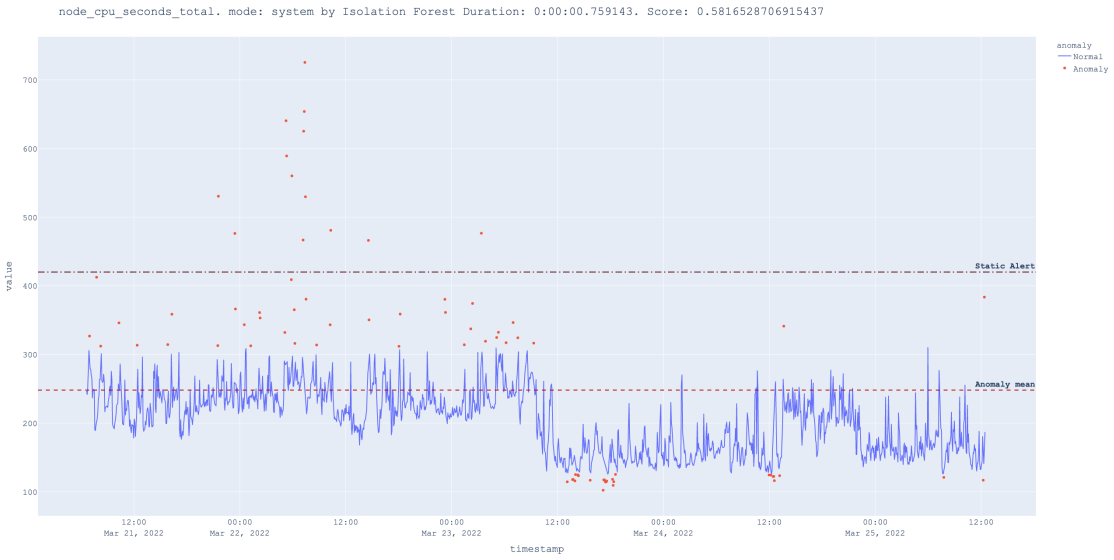


The algorithm recognizes abnormal behavior at points above the static alert line. Nevertheless, the algorithm loses many points since he sees anomalies in a large number of values that are not far from normal behavior but, at the same time, are very far from the line of static alerts.

6.2.4 Isolation Forest

The third algorithm is Isolation Forest, which makes a decision about the anomaly of the value by building forests [14]. Isolation forests were built because anomalies are data points that are ”few and different from each other.” In an isolated forest, randomly selected data is processed based on randomly selected objects in a tree structure. Samples that move deeper into the trees are less likely to be anomalies because more incisions are required to isolate them. Similarly, samples that end with shorter branches indicate anomalies because it was easier for the tree to separate them from other observations 6.4.

Figure 6.4 Isolation Forest

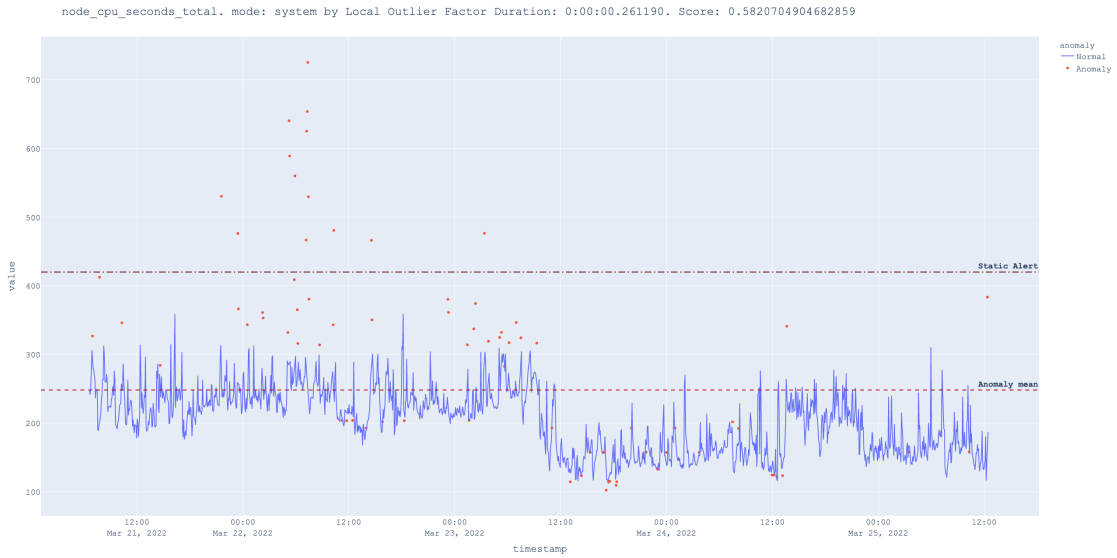


The algorithm is quite clear and really determines even tiny deviations from the norm. However, for monitoring purposes, small deviations are often not important, because many metrics have a large interval of normal behavior.

6.2.5 Local Outlier Factor

The Local Outlier Factor (LOF) algorithm is an unsupervised anomaly detection method that computes the local density deviation of a given data point for its neighbors [15]. The local anomaly factor is based on local density, where locality is given by k nearest neighbors, the distance to which is used to estimate the density. By comparing the local density of an object with the local densities of its neighbors, it is possible to identify areas with a similar density and points that have a significantly lower density than their neighbors. They are considered an anomaly 6.5.

Figure 6.5 Local Outlier Factor

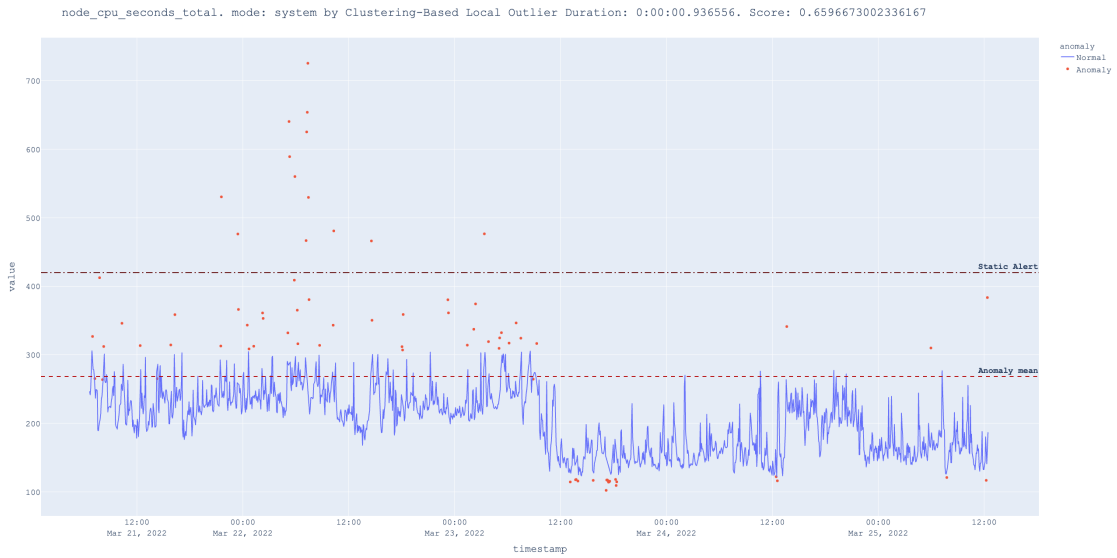


The problem with this algorithm is that the algorithm sees abnormal behavior in less dense data, despite the fact that the values are in the confidence interval.

6.2.6 Clustering-Based Local Outlier

This method uses data partitioning into clusters to detect anomalies. Clustering-based approaches detect outliers by extracting the relationship between objects and the cluster [16]. This outlier detection method assumes that normal data objects belong to large and dense clusters, whereas outliers belong to small or sparse clusters or do not belong to any clusters 6.6.

Figure 6.6 Clustering-Based Local Outlier

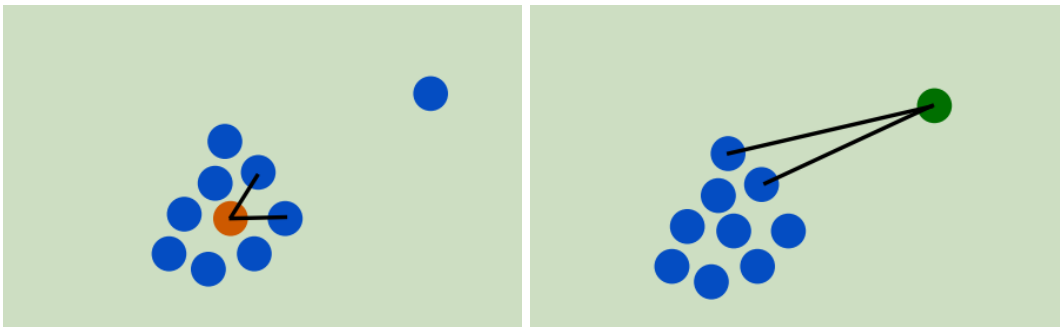


The problem of this algorithm for our purposes is similar to the problem of the previous algo-

rithm. Clustering-Based Local Outlier is too sensitive for data with a large range of acceptable values.

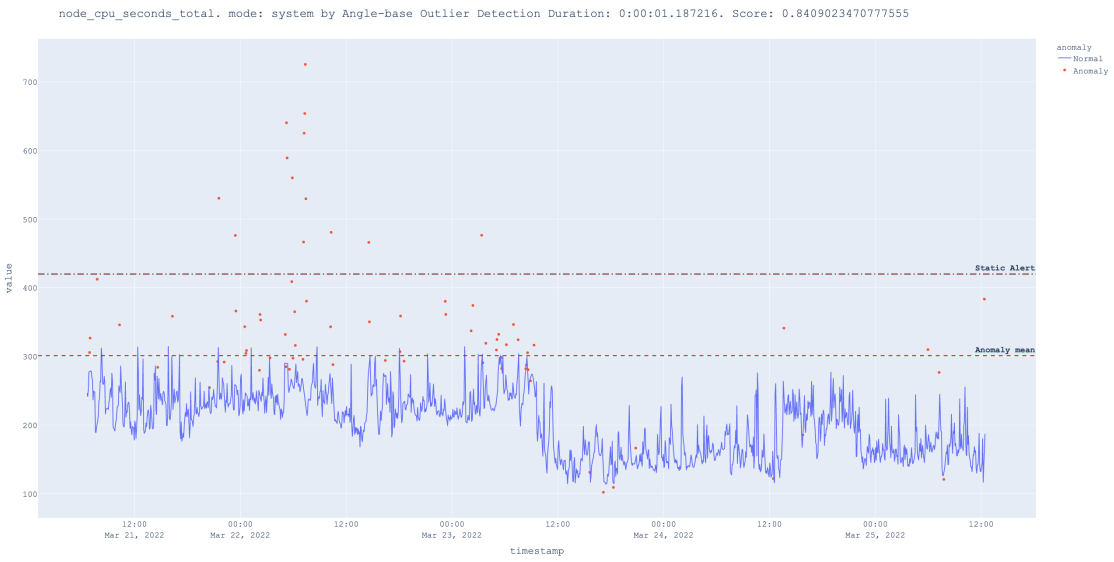
6.2.7 Angle-base Outlier Detection

This algorithm is also not the best choice for our data but is based on a very interesting idea. Angle-based Outlier Detection is based on the idea that for a normal data point, the angle it forms with any two other data points varies greatly as you select different data points [17]. In the case of abnormal values, the angle that a data point forms with other data points does not change much when different data points are selected 6.7.



(a) Normal data point (b) Abnormal data point

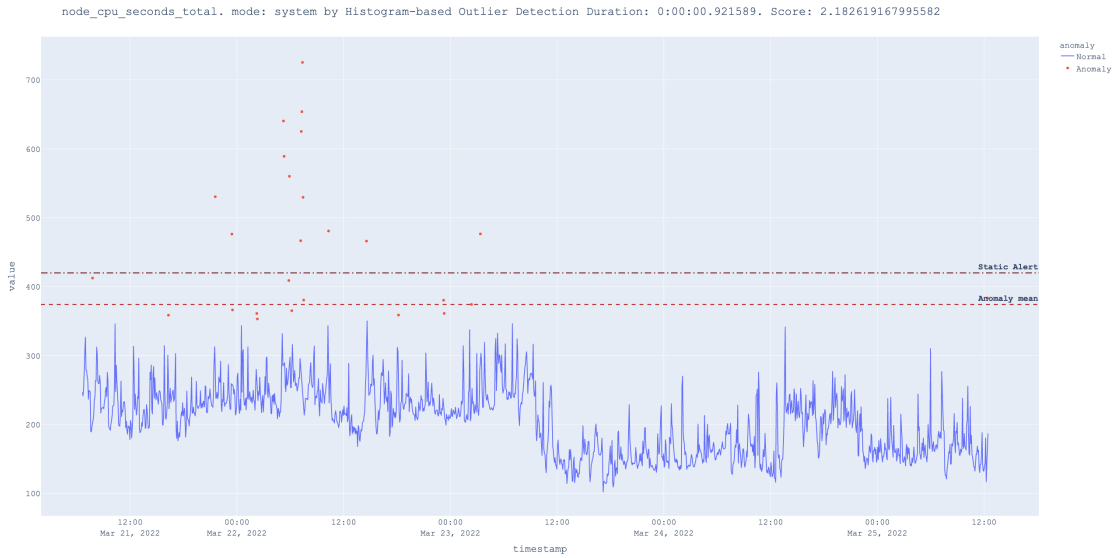
Figure 6.7 Angle-base Outlier Detection



6.2.8 Histogram-based Outlier Detection

The Histogram-based Outlier Detection algorithm is in the top two. The algorithm builds a histogram for each feature, where it estimates the probability of an anomaly for each of them [18]. A data point with a high probability of having an anomaly on several features will be designated as an anomaly 6.8.

Figure 6.8 Histogram-based Outlier Detection



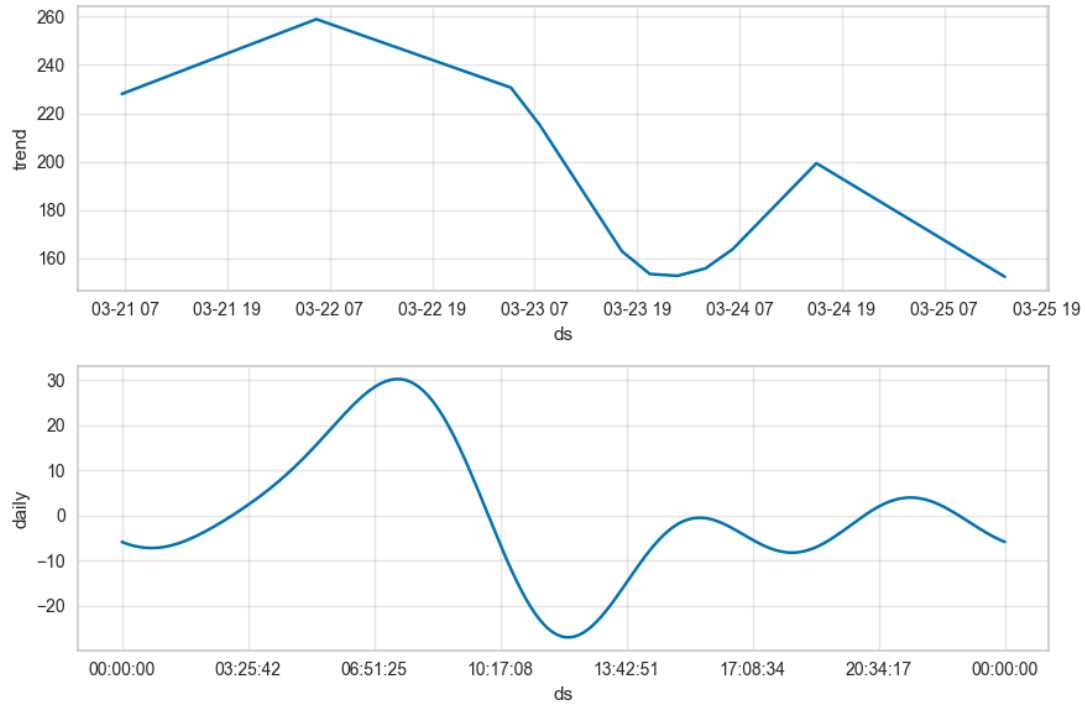
The algorithm shows itself perfectly on our data. Therefore, it has the most significant score by a large margin compared to other algorithms. Histogram-based Outlier Detection does exactly what is required. That is, it detects anomalies in values exceeding the value of the static alert and falsely does not detect normal values. And it is able to warn that the values will probably soon exceed the norm. Considering the above, it was decided to focus on this algorithm and launch it into actual use.

6.2.9 FBprophet

The algorithm from the ”fbprophet” library by Facebook was also tested [9].

The algorithm first outputs patterns and determines trends in data changes depending on time. Specific trends depend on the size of the data. The algorithm is able to make trends during the day, week and even year 6.9.

Figure 6.9 Trends



Here, for example, we can see that the CPU is most loaded during the morning at the beginning of the working day and least at lunch and night.

After that, the algorithm, taking how many date points, predicts the further behavior and the upper and lower edges of the more possible values 6.10. Anomalies are detected depending on the difference in prediction and real values. On the graph, you can see the black points show the real value, the thin blue line is a prediction and the wide blue one is a probable range 6.11.

Figure 6.10 Prediction

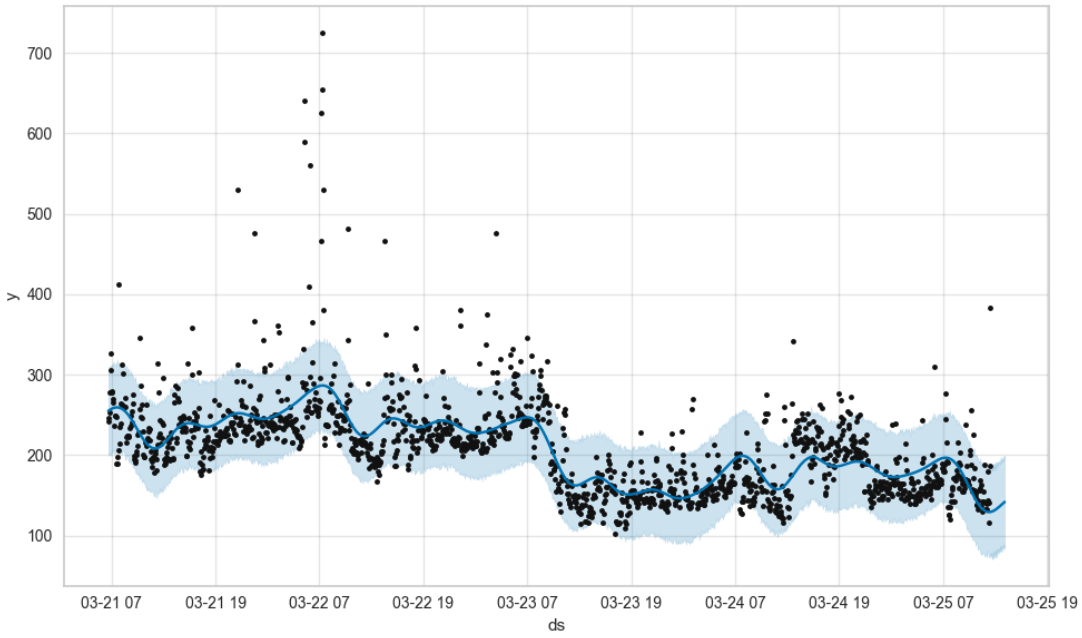
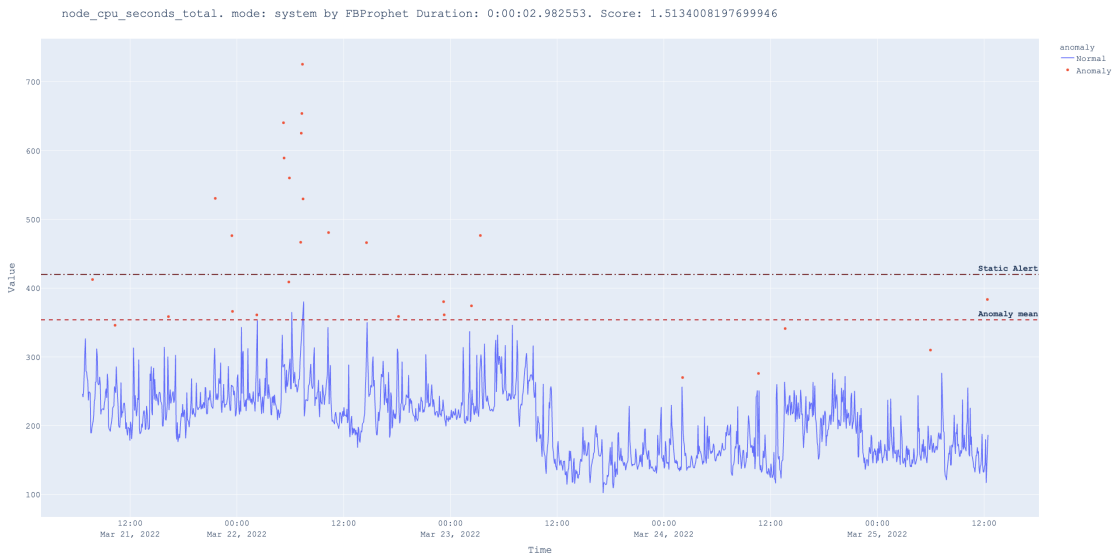


Figure 6.11 FBprophet



6.3 Result of experiment

In repeated tests with different metrics, the situation remained the same. Two leadership contenders have been identified. Which are clearly superior to other algorithms in the quality of anomaly detection 6.12. Histogram-based Outlier Detection was ahead of everyone in terms of detection quality and also detected abnormal behavior 12 hours before the values crossed the

static alert line, as you can see from the graph 6.8. There are plans for the future to conduct a repeat experiment, but with more data for a longer period. I also think that the result may change depending on the hyperparameters of these algorithms. Therefore, in future plans to conduct research with changes in hyperparameters and see how the efficiency of algorithms will change.

Table 6.1 Score table

Score Table	Score	Time
Principal Component Analysis	0	0:00:00.25
Subspace Outlier Detection	0	0:00:04.50
Connectivity-Based Outlier Factor	0.5045	0:00:00.60
Isolation Forest	0.5816	0:00:00.75
Local Outlier Factor	0.5820	0:00:00.26
Clustering-Based Local Outlier	0.6596	0:00:00.93
Angle-base Outlier Detection	0.8409	0:00:01.18
FBprophet	1.5134	0:00:02.98
Histogram-based Outlier Detection	2.1826	0:00:00.92

Figure 6.12 Model score comparison

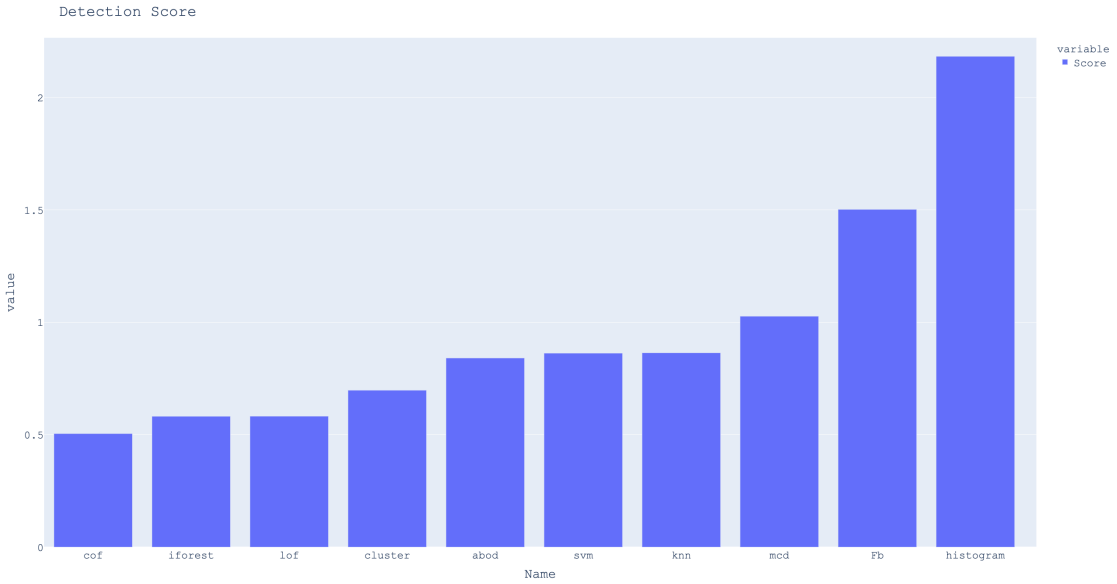
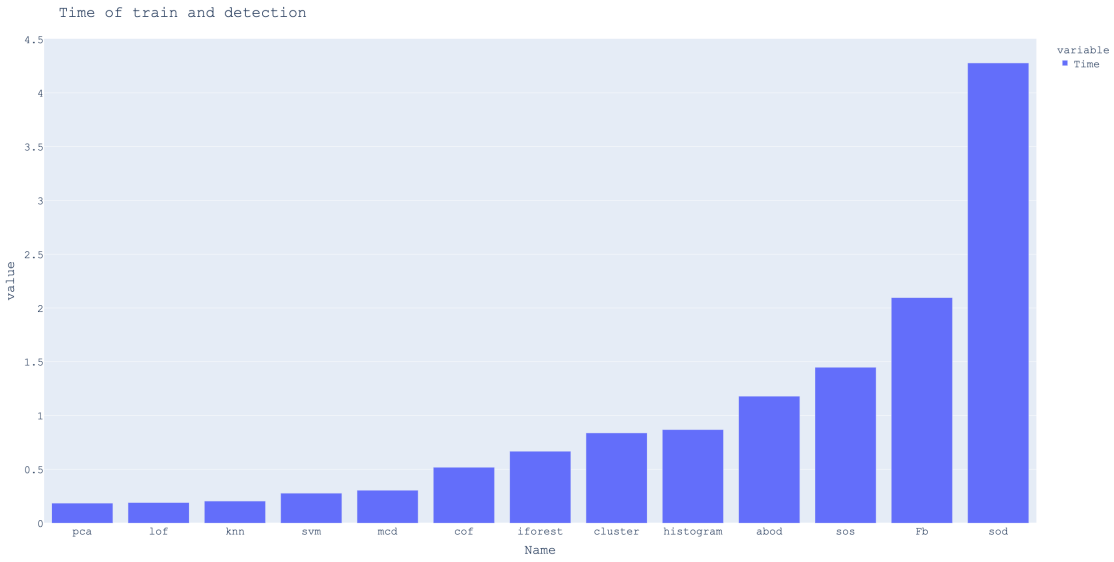


Figure 6.13 Time of training and detection comparison



Chapter 7

Conclusion

As a result, it turned out to make a flexible, automatic data collection and storage system. Where the collection components work independently of each other and are connected using auxiliary scripts such as *AutoDiscovery* and *InfluxParser*. Through which you can configure all monitoring in case of any changes. With the help of scripts, it also turned out to distribute the load on *Prometheuses*, thereby optimizing the system for a larger number of devices. The system calmly copes with monitoring more than 6000 devices. It turned out to build a system of warnings and monitoring of the received data. The system provides the warning with additional information that helps to quickly understand what the problem is and fix it. The anomaly detection system has yet to be finalized by conducting additional experiments with a large amount of data and changing hyperparameters. Also, over time, it will be seen how the algorithms behave after learning on different data.

This project is open-source. [19]

Bibliography

1. EXPORTER, Node. Prometheus/node_exporter: Exporter for machine metrics. *GitHub*. [N.d.]. Available also from: https://github.com/prometheus/node_exporter.
2. PROMETHEUS, howpublished="[online]". Prometheus - Monitoring System, Time Series Database. *Prometheus Blog*. [N.d.]. Available also from: https://prometheus.io/.
3. PROMSCALE. Timescale/promscale: The open-source observability backend for metrics and traces powered by SQL. *GitHub* [[online]]. [N.d.]. Available also from: https://github.com/timescale/promscale.
4. GRAFANA. Grafana/Grafana: The open and composable observability and data visualization platform. visualize metrics, logs, and traces from multiple sources like prometheus, Loki, Elasticsearch, InfluxDB, postgres and many more. *GitHub* [[online]]. [N.d.]. Available also from: https://github.com/grafana/grafana.
5. ALERTMANAGER. Prometheus/Alertmanager: Prometheus Alertmanager. *GitHub* [[online]]. [N.d.]. Available also from: https://github.com/prometheus/alertmanager.
6. Real-time operations: Incident response: On-call. *PagerDuty* [[online]]. 2021. Available also from: https://www.pagerduty.com/.
7. OPSGENIE. Opsgenie: Alerting and on-call management. *opsgenie* [[online]]. [N.d.]. Available also from: https://www.atlassian.com/software/opsgenie.
8. POSTGRESQL, PostgreSQL Global Development. PostgreSQL. *PostgreSQL* [[online]]. 2022. Available also from: https://www.postgresql.org/.
9. FACEBOOK. Quick start. *Prophet* [[online]]. 2022. Available also from: https://facebook.github.io/prophet/docs/quick_start.html.
10. Pycaret. *PyCaret* [[online]]. 2022. Available also from: https://pycaret.org/.
11. JAADI, Zakaria. A step-by-step explanation of principal component analysis (PCA). *Built In*. [N.d.]. Available also from: https://builtin.com/data-science/step-step-explanation-principal-component-analysis.
12. CABERO, Ismael; EPIFANIO, Irene; PIÉROLA, Ana; BALLESTER, Alfredo. Subspace outlier detection. *Knowledge-Based Systems* [[online]]. 2021. Available also from: https://www.sciencedirect.com/science/article/pii/S0950705121000939.
13. WANG, Yanxia; LI, Kang; GAN, Shaojun. A kernel connectivity-based outlier factor algorithm for rare data detection in a baking process. *IFAC-PapersOnLine* [[online]]. 2018. Available also from: https://www.sciencedirect.com/science/article/pii/S2405896318319980.
14. Isolation Forest: Anomaly detection with Isolation Forest. *Analytics Vidhya* [[online]]. 2021. Available also from: https://www.analyticsvidhya.com/blog/2021/07/anomaly-detection-using-isolation-forest-a-complete-guide/.

15. JAYASWAL, Vaibhav. Local outlier factor (LOF). *Medium*. 2020. Available also from: https://towardsdatascience.com/local-outlier-factor-lof-algorithm-for-outlier-identification-8efb887d9843.

16. HE, Zengyou; XU, Xiaofei; DENG, Shengchun. Discovering cluster-based local outliers. *Pattern Recognition Letters* [[online]]. 2003. Available also from: https://www.sciencedirect.com/science/article/pii/S0167865503000035.

17. JAIN, Vishal. Anomaly detection using angle-based techniques. *Medium* [[online]]. 2020. Available also from: https://medium.com/analytics-vidhya/anomaly-detection-using-angle-based-techniques-5d8db8500d14.

18. DATAMAN, Dr. Anomaly detection with histogram-based Outlier Detection (HBO). *Medium* [[online]]. 2021. Available also from: https://medium.com/dataman-in-ai/anomaly-detection-with-histogram-based-outlier-detection-hbo-bc10ef52f23f.

19. IUDIN, Ostap. Seznam.cz a.s. *GitHub* [[online]]. [N.d.]. Available also from: https://github.com/seznam.

Contents of enclosed CD

- src
  - AlertHandler — the directory with AlertHandler 4.2
    - README.md — the file with AlertHandler description
    - configuration — the directory with AlertHandler configuration
    - data — items info and incident history
    - lib — the directory with class and function
    - log — the directory with log of running
    - main.py — the main script file
  - AnomalyDetection — the directory with AnomalyDetection 5
    - configuration — the directory with AnomalyDetection configuration
    - data — metrics files and logs files
    - lib — the directory with class and function
    - main.py — the main script file
  - AutoDiscovery — the directory with AutoDiscovery 2.3
    - configuration — the directory with AutoDiscovery configuration
    - lib — the directory with class and function
    - targets — the directory with targets files
    - main.py — the main script file
  - thesis.pdf — the thesis text in PDF format
  - InfluxDBParser — the directory with InfluxDBParser 3.3
    - config.yaml — the file with InfluxDBParser configuration
    - metric_parser.py — file with functions for metric parsing
    - main.py — the main script file
    - README.md — the file with InfluxDBParser description
  - thesis — the directory with thesis text