Technical report and documentation

DSBD Project: MetricsHub

Mattia Cardillo - **1000037133**

Master's degree student in Computer Engineering

Document code

REVIEW	1.0
CREATION DATE	18/01/2023
LAST REVIEW DATE	28/01/2023

Platform

PREPARED BY	Mattia Cardillo
VERIFIED BY	

Historical reviews

RELEASE

VERS	PREPARED BY		REVIEWED BY	
	Name	**Date**	**Name**	**Date**
1.0	Mattia Cardillo	28/01/2023	Mattia Cardillo	28/01/2023

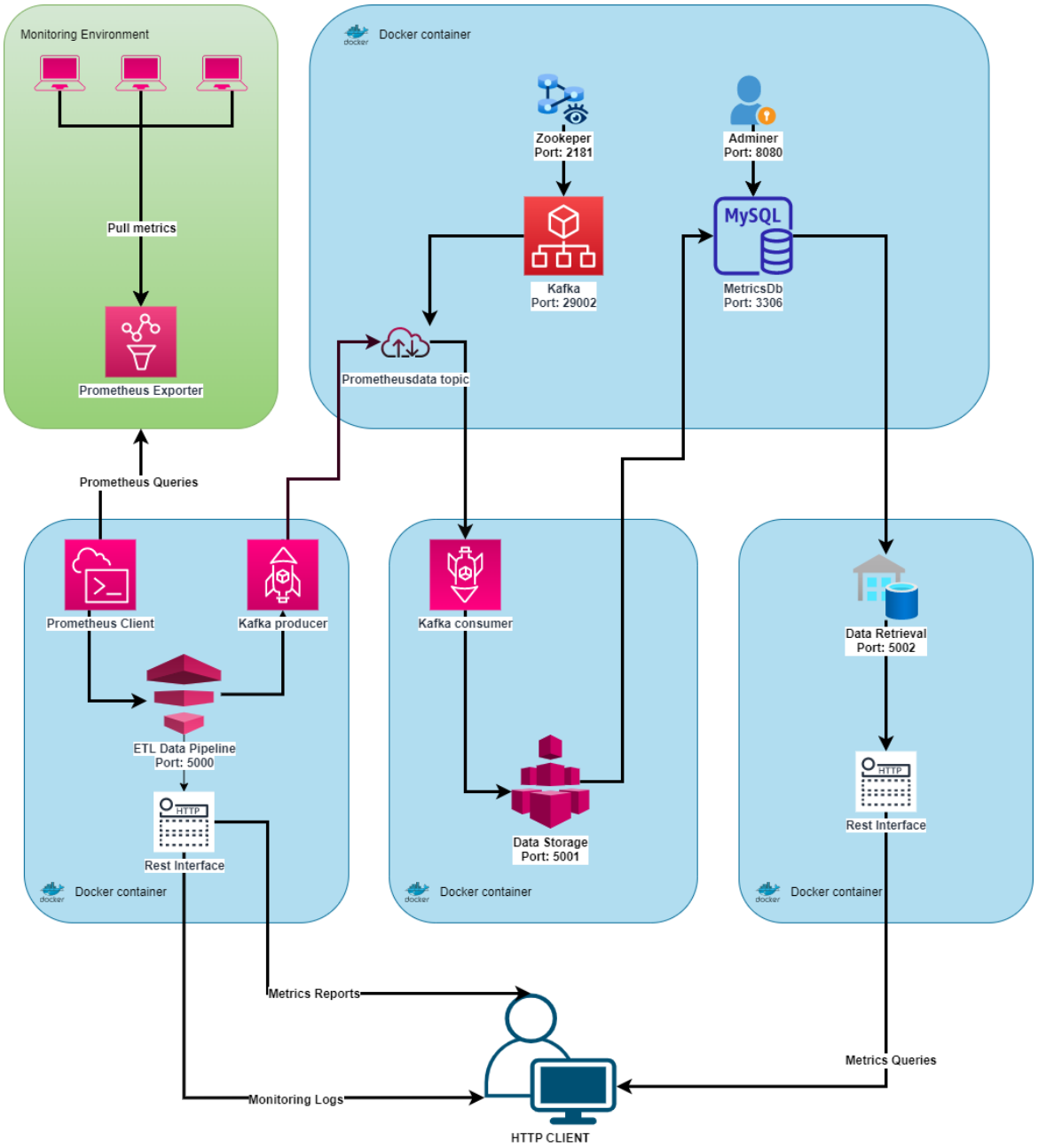
Summary

1. Abstract

MetricsHub is an application that utilizes the microservices architecture to handle data processing, storage and retrieval from a prometheus server. The application consists of three microservices: "ETL Data Pipeline", "Data Storage", and "Data Retrieval". The ETL Data Pipeline microservice is responsible for collecting metrics from a Prometheus server and forwarding them to a Kafka server through a kafka producer. Additionally, it exposes a RESTful interface that provides an easy way to access reports and logs related to the metrics processing. The Data Storage microservice communicates with the Kafka server through a Kafka consumer and performs insertion queries on a MySql database using the data retrieved from Kafka. The Data Retrieval microservice retrieves data from the MySql database and exposes a RESTful interface for external clients to access the data. All microservices are running on Docker, allowing for easy deployment and scaling of the application. By separating the data processing and storage into distinct microservices, the application can handle high volumes of data while maintaining flexibility and scalability.

2. Project Architecture

2.1. Microservices diagram



3. Features

3.1. ETL Data Pipeline

The ETL Data pipeline is a microservice that is responsible for collecting and processing metrics from a Prometheus server. The microservice uses a Prometheus client to retrieve metrics from the server and then processes them using various techniques: for a subset of data, the pipeline calculates stationarity, seasonality, and autocorrelation, for another subset of data, it calculates the mean, minimum, maximum, and standard deviation. These processed metrics are then sent to a Kafka server, where a producer is initialized to produce the message. Additionally, the pipeline generates reports in pdf format that provide detailed information about the processed metrics and produces log files. The service also provides a REST interface that allows users to start again the pipeline process and request log and report files. The ETL Data pipeline is an essential tool for monitoring and analyzing system performance, and it plays a critical role in identifying and addressing potential issues of the monitored systems. The pipeline makes it easy to access and understand the data, providing valuable insights that can help improve the overall performance and reliability of the system. It allows sending the processed metrics to other systems for further analysis and visualization.

3.1.1. Implementation

3.1.1.1. Prometheus Client

A Prometheus client provides an easy way to pull metrics from a Prometheus server in time series format.

```python
from prometheus_api_client.utils import parse_datetime
from datetime import timedelta
from prometheus_api_client import PrometheusConnect, MetricsList, MetricSnapshotDataFrame, MetricRangeDataFrame
from scripts import logsHelpers
import time
import numpy as np

def getAllMetrics(prom, prom_config):
    allJobMetrics = prom.get_current_metric_value(metric_name= '', label_config = prom_config.label_config)
    return allJobMetrics

def getAllMetricsRangeByHour(hour, prom, prom_config, metricList):
    start_time_second = time.time()

    start_time = parse_datetime(str(hour).join("h"))
    end_time = parse_datetime("now")
    chunk_size = timedelta(minutes=5)
    newMetricsArray = []
    newMetricsFormat = {'metric': None, 'max': None, 'min': None, 'avg': None}

    for metricData in metricList:
        print('Running for {}'.format(metricData['metric']['__name__']))
        metric_data = prom.get_metric_range_data(
            metric_name=metricData['metric']['__name__'],
            label_config=prom_config.label_config,
            start_time=start_time,
            end_time=end_time,
            chunk_size=chunk_size,
        )
        metric_df = MetricRangeDataFrame(metric_data)
        max= metric_df['value'].max()
        min= metric_df['value'].min()
        avg= metric_df['value'].mean()
        newMetricsFormat['metric'] = metricData['metric']['__name__']
        newMetricsFormat['max'] = max
        newMetricsFormat['min'] = min
        newMetricsFormat['avg'] = avg
        newMetricsArray.append(newMetricsFormat)

    end_time = time.time()
    duration = end_time - start_time_second
    logsHelpers.getLogger().info('{0}h scripts took {1} seconds to end'.format(hour, duration))

    return newMetricsArray
```

prometheusHelpers.py

3.1.1.2. Metrics retrieval, analytics and report

3.1.1.2.1. Seasonality, Stationarity and Autocorrelation

To perform data analysis, functions and configuration files have been developed to make development maintainable and easy to configure.

```python
def analyzeMetrics():
    print('Start Analyzing Metrics\n')

    for selectedMetric in prom_config.selectedMetrics:
        print('Start test on {}'.format(selectedMetric['name']))
        result = prometheusHelpers.getCustomMetricListFromQuery(prom=prom, hour=48, query=selectedMetric['query'])
        ts = tsManipulationHelpers.parseIntoSeries(result[0]['values'], selectedMetric['name'])

        stationarityResult = tsManipulationHelpers.stationarityTest(ts)
        seasonabilityResult = tsManipulationHelpers.seasonabilityTest(ts, selectedMetric)
        autocorrelationResult = tsManipulationHelpers.autocorrelationTest(ts)

        reportsHelpers.writeReport(selectedMetric['name'], ['Test di stazionarietà:', stationarityResult, 'Test di stagionalità:', seasonabilityResult,

    print('End of the first step \n')
    return
```

analyze Metrics function in analyticsHelpers.py

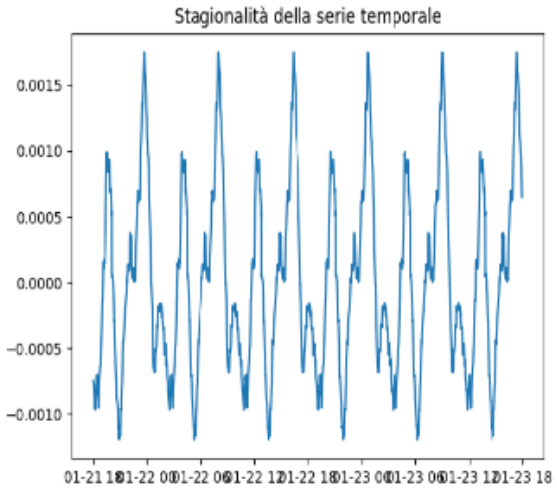
```python
prometheushostname="http://15.160.61.227:29090"

label_config = {'job': 'host'}


selectedMetrics = [
    {
     'name': 'promhttp_metric_handler_requests_total',
     'query': '{job="host",__name__="promhttp_metric_handler_requests_total", code="200"}',
     'params': {
         'period': 350,
         'model': 'additive'
     }
    },
    {
     'name': 'pushgateway_http_push_duration_seconds',
     'query': '{job="collector", __name__="pushgateway_http_push_duration_seconds", quantile="0.5", method="put"}',
     'params': {
         'period': 500,
         'model': 'additive'
     }
    }
]
```

prometheus.py in config directory

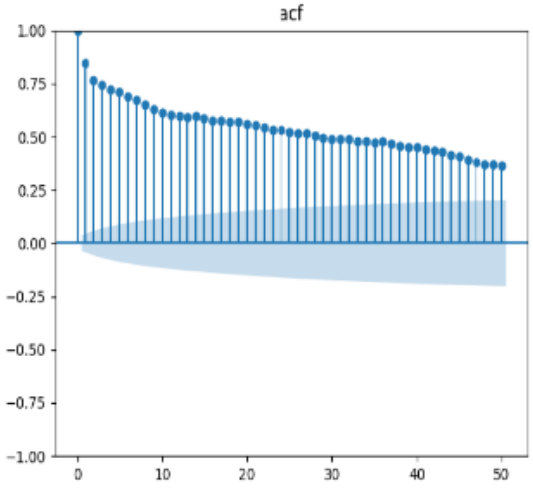
For this study case, metrics "promhttp_metric_handler_requests_total" and "pushgateway_http_push_duration_seconds" were evaluated.

- Metric "pushgateway_http_push_duration_seconds" was chosen as it is typically a stationary and autocorrelated metric, measuring the number of HTTP requests sent.
  This metric may be considered stationary if HTTP requests to the system are evenly distributed over time and if the system is not undergoing significant changes, moreover it could be considered autocorrelated if the response times of HTTP requests are correlated with each other. For example, if one HTTP request takes longer to process, it is likely that subsequent requests will also take longer.
  The microservice, after processing and analyzing the metric, has produced the following pdf report, which shows that as expected, the metric is stationary and autocorrelated:

Test di stazionarietà:
La serie temporale è stazionaria
Test di stagionalità:



Test di autocorrelazione:

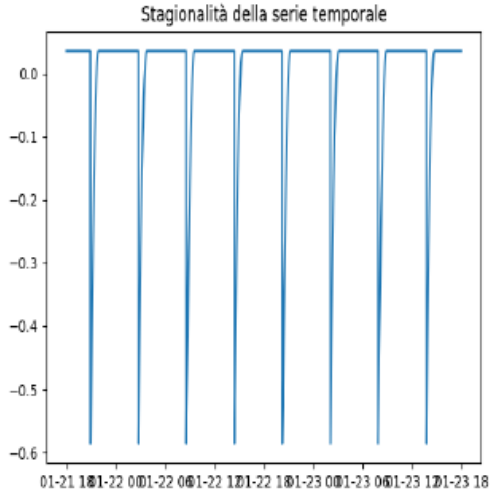


pushgateway_http_push_duration_seconds.pdf

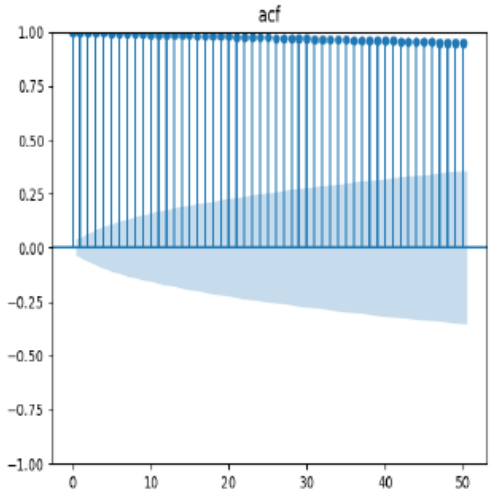
- Metric "promhttp_metric_handler_requests_total" was selected as it is expected to exhibit seasonality.
  This metric may be considered seasonal if the number of requests to the system follows a predictable pattern over time, such as an increase in requests during certain times of the day, week, or month.

  The microservice, after processing and analyzing the metric, has produced the following pdf report, which shows that as expected, the metric is seasonal.

Test di stazionarietà:
La serie temporale non è stazionaria
Test di stagionalità:



Test di autocorrelazione:



promhttp_metric_handler_requests_total.pdf

3.1.1.2.2. Max, min, avg and std_dev

To perform data processing and data analysis, functions and configuration files have been developed to make development maintainable and easy to configure.

```
def retrieveData():
    print('Start Data Retrieval \n')

    calculatedValues = {
        'metrics1hData': prometheusHelpers.getCustomMetricsRangeByHour(prom=prom, prom_config=prom_config, hour=1, metricName="go.*"),
        'metrics3hData': prometheusHelpers.getCustomMetricsRangeByHour(prom=prom, prom_config=prom_config, hour=3, metricName="go.*"),
        'metrics12hData': prometheusHelpers.getCustomMetricsRangeByHour(prom=prom, prom_config=prom_config, hour=12, metricName="go.*"),
    }

    print('End of the second step\n')
    return calculatedValues
```

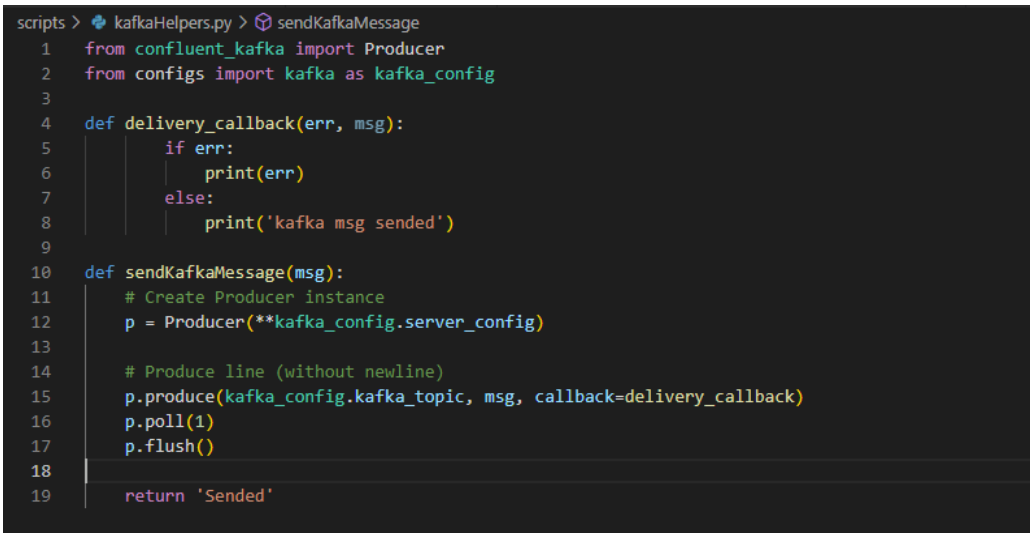
retrieveData function in analyticsHelpers.py

3.1.1.3. Kafka Producer

Kafka handles data streams and communication between microservices in a distributed architecture.

This code creates an instance of a Kafka producer using the settings specified in "kafka_config.server_config". Then it sends a message specified by "msg" to the topic specified in "kafka_config.kafka_topic" using the "produce" method of the producer. The "poll (1)" and "flush" methods are used to ensure that the message is

actually sent to the topic. "delivery_callback" is a function that can be used to handle the delivery confirmation of the message.

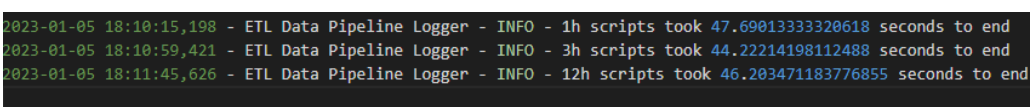


```python
scripts > kafkaHelpers.py > sendKafkaMessage
from confluent_kafka import Producer
from configs import kafka as kafka_config

def delivery_callback(err, msg):
        if err:
            print(err)
        else:
            print('kafka msg sended')

def sendKafkaMessage(msg):
    # Create Producer instance
    p = Producer(**kafka_config.server_config)

    # Produce line (without newline)
    p.produce(kafka_config.kafka_topic, msg, callback=delivery_callback)
    p.poll(1)
    p.flush()

    return 'Sended'
```

kafkaHelpers.py

3.1.1.4. Logs

An internal monitoring system was implemented to log the execution time of various features and provide a RESTful interface for accessing the log file



```
2023-01-05 18:10:15,198 - ETL Data Pipeline Logger - INFO - 1h scripts took 47.69013333320618 seconds to end
2023-01-05 18:10:59,421 - ETL Data Pipeline Logger - INFO - 3h scripts took 44.22214198112488 seconds to end
2023-01-05 18:11:45,626 - ETL Data Pipeline Logger - INFO - 12h scripts took 46.203471183776855 seconds to end
```

05_01_2023.log

3.1.2. API list

3.1.2.1. API “Ping”

3.1.2.1.1. Overview

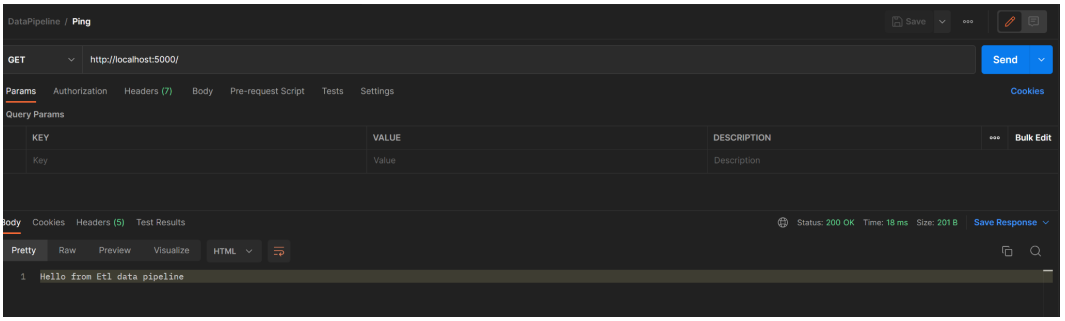
This API allows the client to test the online status of the microservice

3.1.2.1.2. Request URL and method

The flow must be carried out with the `GET` method.

The URL to contact will be:

`https://{{host_server}}/`



3.1.2.2. API “Restart”

3.1.2.2.1. Overview

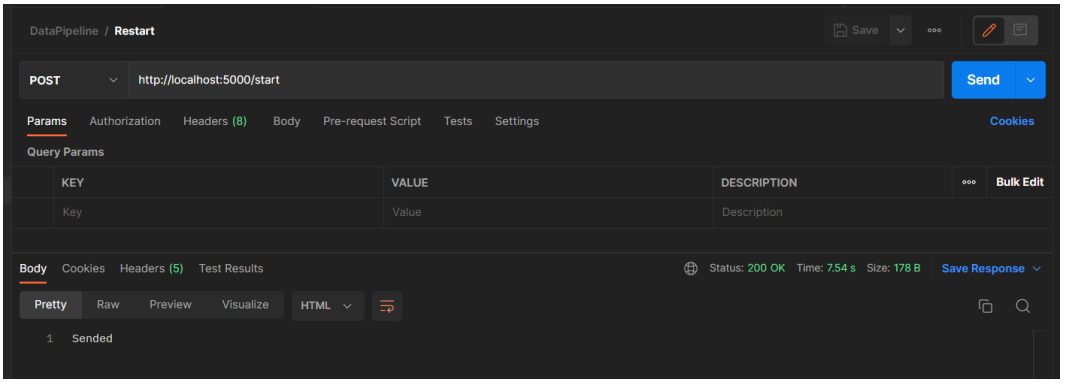
This API allows the client to restart the pipeline process, from prometheus queries to kafka message sending.

3.1.2.2.2. Request URL and method

The flow must be carried out with the `POST` method.

The URL to contact will be:

`https://{{host_server}}/start`



3.1.2.3. API “Logs”

3.1.2.3.1. Overview

This API allows the client to retrieve logs file ordered by day, that contains indication of how much time the process spent to make metrics computing.

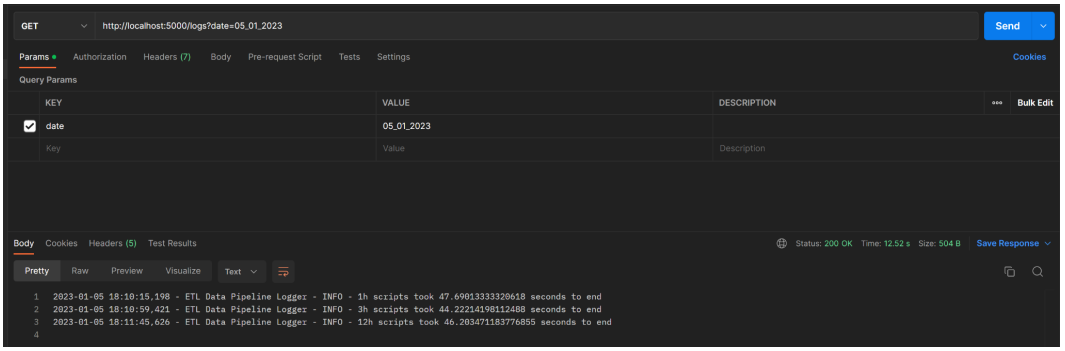
3.1.2.3.2. Request URL and method

The flow must be carried out with the `GET` method.

The URL to contact will be:

`https://{{host_server}}/logs/{{formatted_date}}`

formatted_date = dd_mm_yyyy



3.1.2.4. API “Reports”

3.1.2.4.1. Overview

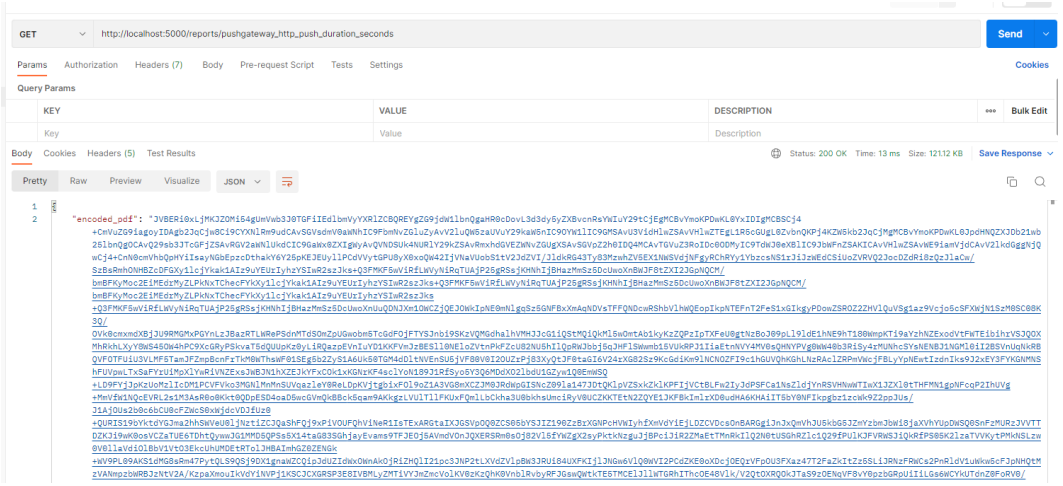
This API allows the client to retrieve reports files as a base64 string.

3.1.2.4.2. Request URL and method

The flow must be carried out with the `GET` method.

The URL to contact will be:

`https://{{host_server}}/reports/{{name}}`



3.2. Data Storage

The Data Storage microservice is responsible for retrieving and storing data from a Kafka server. It initializes a consumer that subscribes to the "prometheusdata" topic and retrieves messages from the server. The service then transforms the message into an object and stores the data in a MySQL database. This microservice plays a critical role in the data flow by ensuring that the data is properly stored and can be easily accessed for further analysis and visualization. It allows final users to have a data history that can be useful for monitoring and troubleshooting.

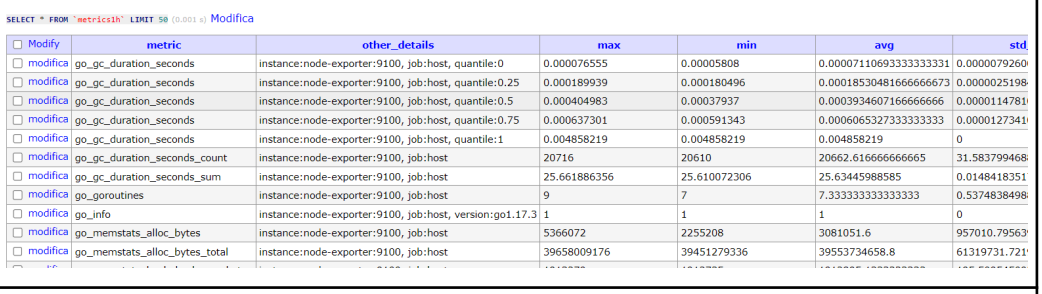
3.2.1. Implementation

3.2.1.1. Kafka Consumer and Database Client

A Kafka consumer is created using the configuration specified in "kafka_configs.consumer_config" and subscribes to a topic specified in "kafka_configs.topic_name". It then enters a loop where it polls for messages from the topic using the "poll" method of the consumer. If a message is received, it is decoded and printed to the console. If the message is in json format, it extracts the metrics in 3 different data sets, metrics1hData, metrics3hData and metrics12hData. It then uses data from the extracted metrics and makes queries to the database using the "makeQuery" method from dbHelpers, passing the queries and the data as parameters. The queries are specified in the "queries" field of the dbHelpers object. If an error occurs, it is printed to the console. The consumer is closed when the loop is exited.

```
def startConsumeKafka():
    print('Consumer Start')
    c = Consumer(kafka_configs.consumer_config)
    c.subscribe([kafka_configs.topic_name])

    try:
        while (True):
            msg = c.poll(1.0)
            if msg is None:
                continue
            if msg.error():
                print("Consumer error: {}".format(msg.error()))
                continue
            metricMsg = msg.value().decode('utf-8')
            print('Received message: {}'.format(metricMsg))
            metricMsg = json.loads(msg.value().decode('utf-8'))
            for msg in metricMsg["metrics1hData"]:
                dbHelpers.makeQuery(dbHelpers.queries["insertMetric1h"], (msg["metric"], msg["max"], msg["min"], msg["avg"], msg['std_dev']))

            for msg in metricMsg["metrics3hData"]:
                dbHelpers.makeQuery(dbHelpers.queries["insertMetric3h"], (msg["metric"], msg["max"], msg["min"], msg["avg"], msg['std_dev']))

            for msg in metricMsg["metrics12hData"]:
                dbHelpers.makeQuery(dbHelpers.queries["insertMetric12h"], (msg["metric"], msg["max"], msg["min"], msg["avg"], msg['std_dev']))
        c.close()
    except Exception as e:
        print(e)

    return
```

startConsumeKafka function in kafkaHelpers.py



SELECT * FROM `metrics1h` LIMIT 50 (0.001 s) Modifica

Modify	metric	other_details	max	min	avg	std
modifica	go_gc_duration_seconds	instance:node-exporter:9100, job:host, quantile:0	0.000076555	0.00005808	0.00007110693333333331	0.0000079260
modifica	go_gc_duration_seconds	instance:node-exporter:9100, job:host, quantile:0.25	0.000189939	0.000180496	0.00018530481666666673	0.0000025198
modifica	go_gc_duration_seconds	instance:node-exporter:9100, job:host, quantile:0.5	0.000404983	0.00037937	0.0003934607166666666	0.0000114781
modifica	go_gc_duration_seconds	instance:node-exporter:9100, job:host, quantile:0.75	0.000637301	0.000591343	0.0006065327333333333	0.0000127341
modifica	go_gc_duration_seconds	instance:node-exporter:9100, job:host, quantile:1	0.004858219	0.004858219	0.004858219	0
modifica	go_gc_duration_seconds_count	instance:node-exporter:9100, job:host	20716	20610	20662.616666666665	31.583799468
modifica	go_gc_duration_seconds_sum	instance:node-exporter:9100, job:host	25.661886356	25.610072306	25.63445988585	0.0148418351
modifica	go_goroutines	instance:node-exporter:9100, job:host	9	7	7.333333333333333	0.5374838498
modifica	go_info	instance:node-exporter:9100, job:host, version:go1.17.3	1	1	1	0
modifica	go_memstats_alloc_bytes	instance:node-exporter:9100, job:host	5366072	2255208	3081051.6	957010.79563
modifica	go_memstats_alloc_bytes_total	instance:node-exporter:9100, job:host	39658009176	39451279336	39553734658.8	61319731.721

metrics1h table view after storage process

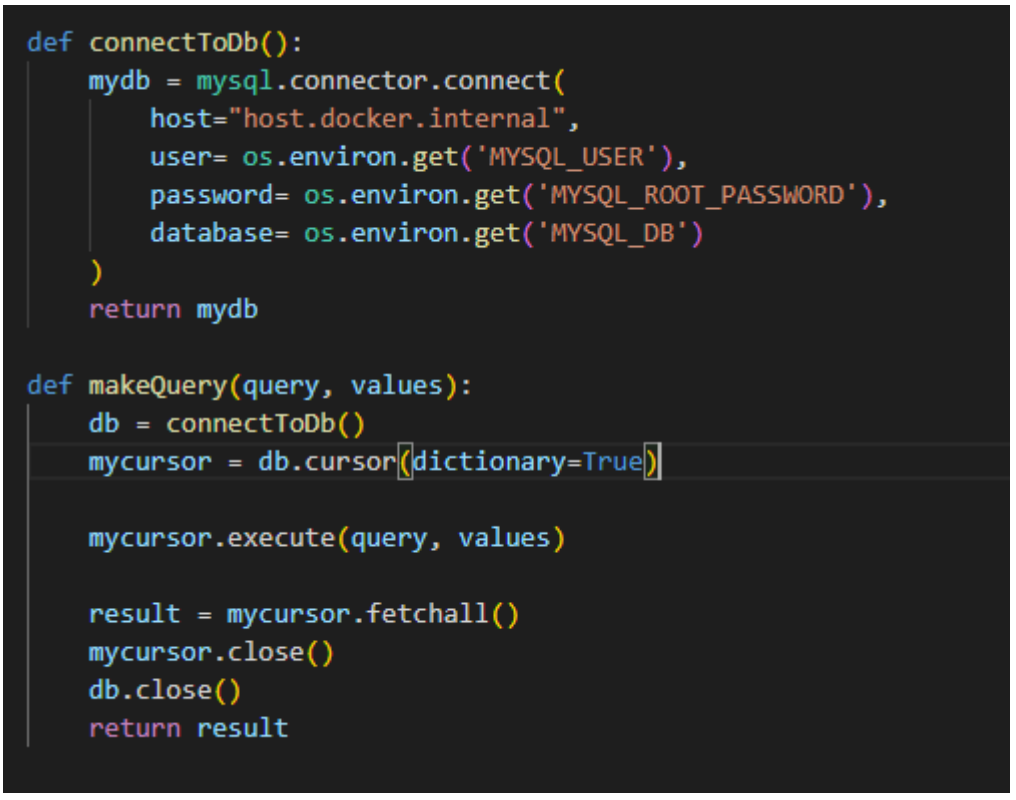
3.3. Data Retrieval

The Data Retrieval microservice provides an easy way to access historical metrics stored in a MySQL database through a RESTful interface. It allows clients to query the database for specific metrics by name, max, min, std_dev, and time range. The service utilizes the creation of views to make it easier for clients to retrieve the metrics they need. This microservice enables users to easily access and analyze stored data, providing valuable insights that can help improve the overall performance and reliability of the system. It also allows a powerful and flexible way to retrieve the data that a client needs to analyze.

3.3.1. Implementation

3.3.1.1. Database client

The "connectToDb" function creates a connection to a MySQL database using the "mysql.connector.connect" method. It connects to the database specified by environment variables "MYSQL_USER", "MYSQL_ROOT_PASSWORD" and "MYSQL_DB" and returns the connection object. The "makeQuery" function takes two parameters, a query and a tuple of values. It calls the "connectToDb" function to get the database connection and creates a cursor object. It then executes the query using the values passed in and retrieves the result using "fetchall". The cursor and the connection are closed before returning the result.



```python
def connectToDb():
    mydb = mysql.connector.connect(
        host="host.docker.internal",
        user= os.environ.get('MYSQL_USER'),
        password= os.environ.get('MYSQL_ROOT_PASSWORD'),
        database= os.environ.get('MYSQL_DB')
    )
    return mydb

def makeQuery(query, values):
    db = connectToDb()
    mycursor = db.cursor(dictionary=True)

    mycursor.execute(query, values)

    result = mycursor.fetchall()
    mycursor.close()
    db.close()
    return result
```

dbHelpers.py

3.3.1.2. Rest Interface

3.3.2. API list

3.3.2.1. API “Ping”

3.3.2.1.1. Overview

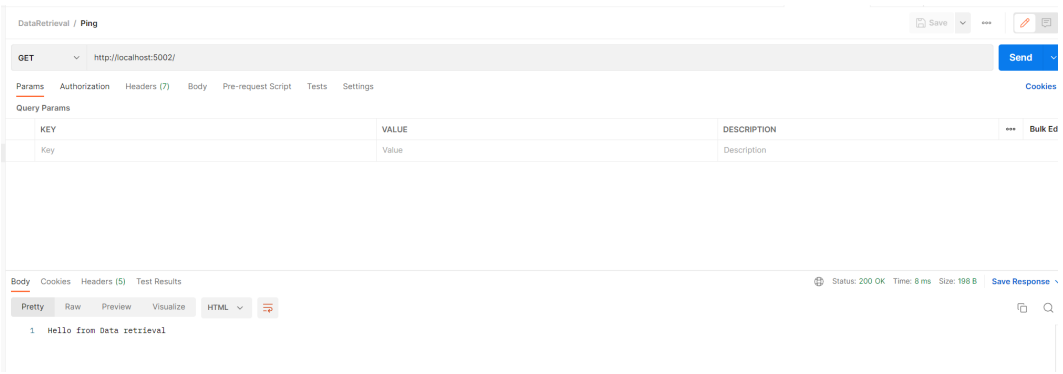
This API allows the client to test the online status of the microservice

3.3.2.1.2. Request URL and method

The flow must be carried out with the `GET` method.

The URL to contact will be:

`https://{{host_server}}/`



3.3.2.2. API “getMetricsNames”

3.3.2.3. Overview

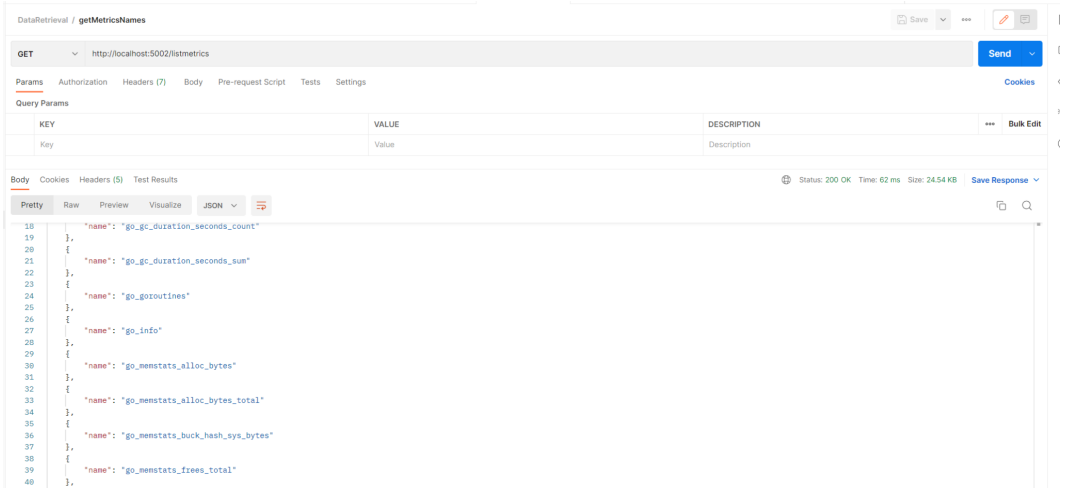
This API allows the client to request all names of the metrics stored in the db.

3.3.2.4. Request URL and method

The flow must be carried out with the `GET` method.

The URL to contact will be:

`https://{{host_server}}/listmetrics`



3.3.2.5. API “getMetrics”

3.3.2.6. Overview

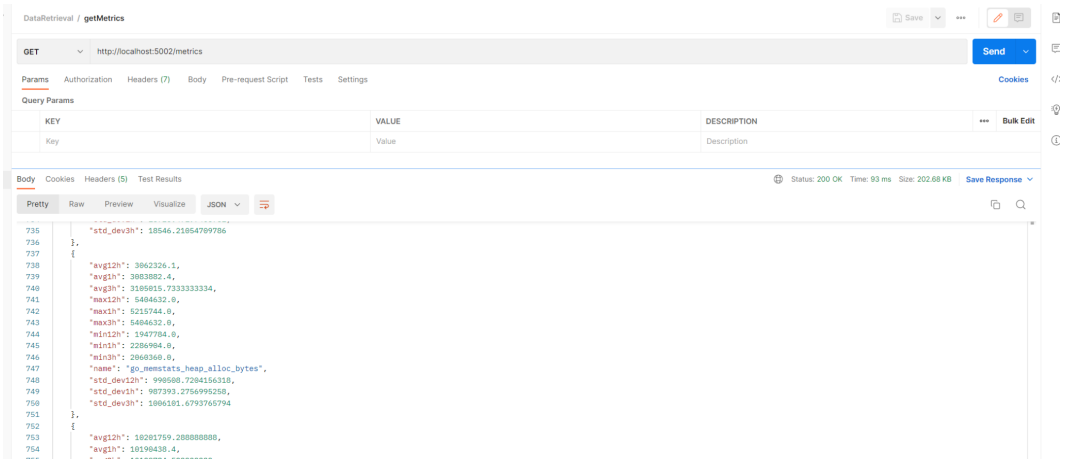
This API allows the client to retrieve all data like avg. max, min, std_dev of the metrics stored in the db.

3.3.2.7. Request URL and method

The flow must be carried out with the `GET` method.

The URL to contact will be:

`https://{{host_server}}/metrics`



3.3.2.8. API “getMetricsFiltered”

3.3.2.9. Overview

This API allows the client to retrieve avg, max, min, std_dev of a metrics with range filters.

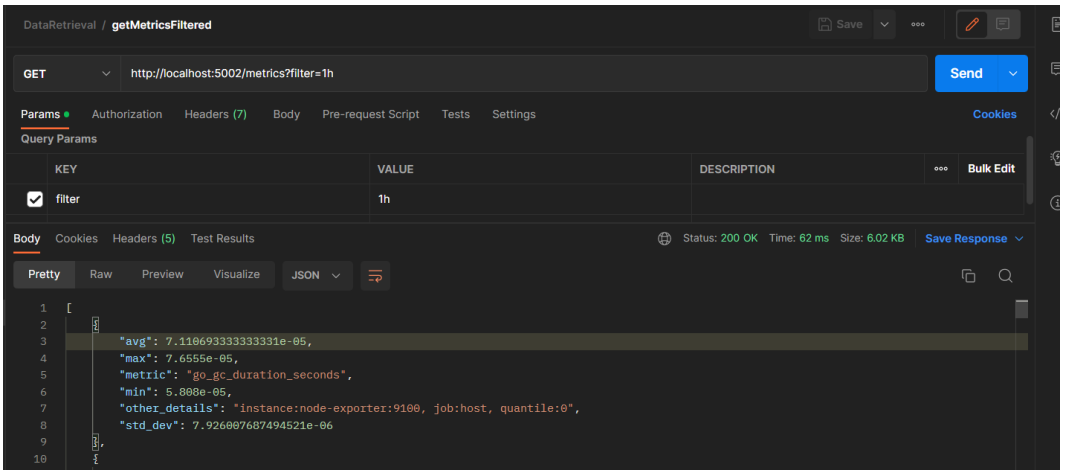
A range filter can be added using “filter” query param in /metrics api.

3.3.2.10. Request URL and method

The flow must be carried out with the `GET` method.

The URL to contact will be:

`https://{{host_server}}/metrics?filter={{filter}}`



3.3.2.11. API “getMetricsByName”

3.3.2.12. Overview

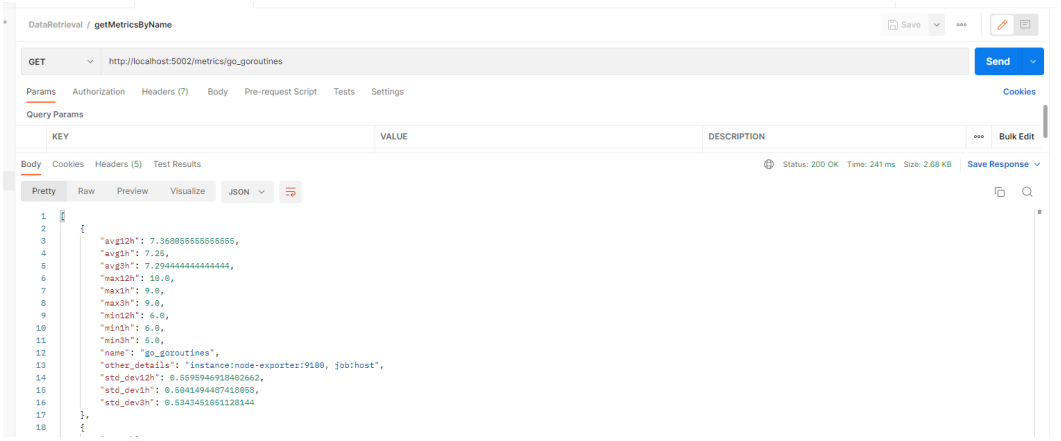
This API allows the client to retrieve avg. max, min, std_dev filtering by metrics name.

3.3.2.13. Request URL and method

The flow must be carried out with the `GET` method.

The URL to contact will be:

`https://{{host_server}}/metrics/{{metric_name}}`



3.4. Postman

All the features can be easily tested using an HTTP client like postman.

Just import the collections located in “postman_collections” folder

4. Requirements and installation

The requirements listed in this document will guide the development of the system and will be used to evaluate the success of the project. This document will be updated as the project progresses, and new requirements may be added as necessary.

To run the project locally:

- Install Python 3.7 and the libraries:
    - flask
    - confluent_kafka

    - datetime
    - prometheus_api_client
    - statsmodels
    - reportlab
    - mysql-connector-python
    - python-dotenv
- run docker-compose-up on docker-compose.yml in the root of the project to start Kafka, Zookeper, MySql and Adminer
- use the command “python main.py” on etldatapipeline folder to start the microservice
- use the command “python main.py” on datastorage folder to start the microservice
- use the command “python main.py” on dataretrieval folder to start the microservice
- use a client like postman to watch the results

To run the project in docker environment:

- Use the docker-compose up on every docker container, particularly:
    - run docker-compose-up on docker-compose.yml located in the root of the project to start Kafka, Zookeper, MySql and adminer
    - run docker-compose-up on docker-compose.yml located in the etldatapipeline folder to start installing requirements and start the etldatapipeline application on port 5000

- run docker-compose-up on docker-compose.yml located in the datastorage folder to start installing requirements and start the datastorage application on port 5001
- run docker-compose-up on docker-compose.yml located in the dataretrieval folder to start installing requirements and start the dataretrieval application on port 5002
- use a client like postman to watch the results