



**UNIVERSITÀ
degli STUDI
di CATANIA**

Corso di Distributed Systems and Big Data– Anno Accademico 2022/2023

Docente: Antonella Di Stefano

Studenti:

Mario Gabriele Sambataro (1000042297), Rapisarda Gabriele(1000042319)

**Progetto prova in itinere
DSBD**

Sommario

Introduzione	3
Diagramma dei micro-servizi	4
mySQL	5
Zoo e KAFKA	5
Asynchronous messaging:	6
Init-kafka	6
ETL data pipeline	6
Scraping dati prometheus	7
Elaborazione dati	7
Data Storage	8
Connessione al database	8
Comportamento microservizio.....	8
Data Retrieval	9
GET API	10
SLA Manager	11
TALEND API TESTER	11
Post API -> ETL data pipeline	12
GET API	13

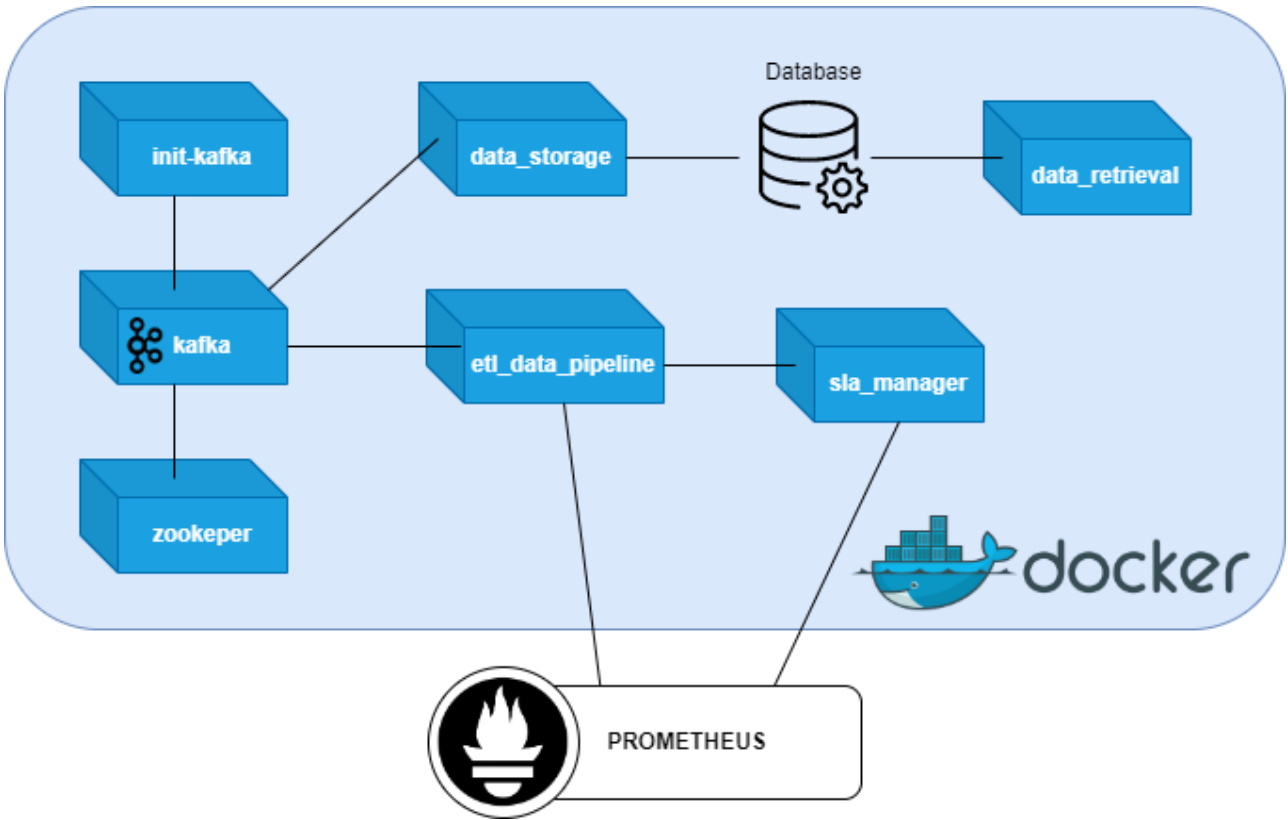
Introduzione

Lo sviluppo dell'applicazione è avvenuto sfruttando le funzionalità fornite dall'ambiente Docker per creare un applicativo in grado di gestire diversi microservizi e di elaborare dati provenienti da un server Prometheus.

In particolare, sono stati implementati i seguenti container:

- **MySQL(db):** questo container mantiene una distribuzione MySQL, attualmente nella versione 8.0.32. Per fare in modo che i dati vengano memorizzati in maniera persistente (così da evitare che vadano perduti a seguito di aggiornamenti o modifiche al codice).
- **Zoo e Kafka:** i due container conterranno i rispettivi servizi necessari ad implementare il messaging asincrono tipico della piattaforma Apache Kafka.
- **Init-Kafka:** container che permette di inizializzare il topic utilizzato per implementare la comunicazione asincrona.
- **ETL data pipeline:** come da definizione questo microservizio prevede una serie di step di processamento di dati provenienti da una specificata applicazione, nel caso in esame si è utilizzata come sorgente il seguente server Prometheus <http://15.160.61.227:29090/>. Verranno valutate un set di 5 metriche negli intervalli di tempo di 1h, 3h e 12h.
- **Data Storage:** microservizio che tramite Kafka preleva i dati inviati sullo specifico topic, e li inserisce nelle apposite tabelle del DB "DataStorage" creato tramite il container mysql.
- **Data Retrieval:** microservizio che offre un interfaccia REST API in grado di effettuare QUERY sul DB sopracitato e restituire attraverso delle GET le informazioni precedentemente processate dall' ETL data pipeline e salvate tramite l'apposito microservizio.
- **SLA Manager:** microservizio che permette di definire uno SLA set composto da 5 metriche (tali metriche vengono ricavate tramite una POST API effettuata sfruttando l'estensione TALEND API TESTER), che restituisce attraverso delle GET il numero di violazioni passate e previste nei prossimi 10m (numero di volte che il parametro sotto esame sfiora un range di valori inviato al microservizio tramite la sopracitata POST API).

Diagramma dei micro-servizi



mySQL

```
db:
  image: mysql
  restart: always
  ports:
    - 3306:3306
  environment:
    MYSQL_DATABASE: DataStorage
    MYSQL_USER: user
    MYSQL_PASSWORD: pswd
    MYSQL_ROOT_PASSWORD: pswd
  volumes:
    - ./data_storage/database.sql:/docker-entrypoint-initdb.d/database.sql
  container_name: db
```

docker.compose.yml --mySQL

Il seguente container è stato inizializzato con i settaggi sopra mostrati, inoltre tramite il file database.sql il database usato per salvare i dati processati viene opportunamente inizializzato:

1. viene creato il DB “DataStorage” se non è presente
2. Vengono create le tabelle:
 - metrics
 - metadata
 - prediction

```
data_storage > database.sql
1  CREATE DATABASE IF NOT EXISTS DataStorage;
2  USE DataStorage;
3
4  CREATE TABLE IF NOT EXISTS metadata (
5    ID int AUTO_INCREMENT, metric_name varchar(255),
6    autocorrelation varchar(255),
7    stationarity varchar(255),
8    seasonality varchar(5000) ,
9    PRIMARY KEY (ID));
10
11 CREATE TABLE IF NOT EXISTS prediction (
12   ID int AUTO_INCREMENT,
13   metric_name varchar(255),
14   max double,
15   min double,
16   avarage double,
17   PRIMARY KEY (ID));
18
19 CREATE TABLE IF NOT EXISTS metrics (
20   ID int AUTO_INCREMENT,
21   metric_name varchar(255),
22   start_time varchar(255),
23   max double,
24   min double,
25   avarage double,
26   std double,
27   PRIMARY KEY (ID));
```

database.sql

Zoo e KAFKA

La comunicazione tra i vari micro-servizi dell’applicativo avviene tramite **Kafka**, che fungerà da sistema di messaging asincrono. In particolare, il nostro micro-servizio “ETL data pipeline” sarà il **produttore**, mentre il microservizio “data storage” come il **consumatore**.

```
data_storage > main.py > ...
10  #Data_storage
11
12  try:
13      c=Consumer({
14          'bootstrap.servers': 'kafka:29092',
15          'group.id': 'mygroup',
16          'auto.offset.reset': 'latest'
17      })
18      c.subscribe(['prometheusdata'])
19  except Exception:
20      print(Exception)
21
```

Settings Kafka Consumer

```
etl_data_pipeline > main.py > sendtokafka
120
121
122 def sendtokafka(fileJson,key_file):
123
124     #EtL_data_pipeline
125
126     broker= "kafka:29092"
127     topic="prometheusdata"
128     conf={'bootstrap.servers':broker}
129     p=Producer(**conf)
130
131     def delivery_callback(err, msg):
132         if err:
133             sys.stderr.write('%s Message failed delivery: %s\n' % err)
134         else:
135             sys.stderr.write('%s Message delivered to %s [%d] @ %d\n' %
136                             (msg.topic(), msg.partition(), msg.offset()))
137
138     try:
139         value= json.dumps(fileJson)
140
141         p.produce(topic, key = key_file, value=value, callback=delivery_callback)
142     except BufferError:
143         sys.stderr.write(' Errori \n' )
144
145     p.poll(0)
146     p.flush()
```

Settings Kafka Producer

Asynchronous messaging:

La comunicazione tra i due microservizi è stata gestita nel seguente modo:

- **data storage - consumer:**
Una volta effettuata la subscription al topic “prometheusdata” si metterà in ascolto di eventuali messaggi provenienti dal broker.
- **ETL data pipeline - producer:**
Una volta effettuata la connessione al broker, il microservizio sfruttando la *produce* invierà i dati strutturati, identificandone la tipologia attraverso una chiave, che verrà utilizzata dal data storage per discriminare il contenuto del messaggio ed effettuare la corretta operazione di salvataggio sul DB.

```
p.produce(topic, key = key_file, value=value, callback=delivery_callback)
```

Settings produce

Init-kafka

```
init-kafka:
  image: confluentinc/cp-kafka:latest
  depends_on:
    - kafka
  entrypoint: [ '/bin/sh', '-c' ]
  command: |
    "
    # blocks until kafka is reachable
    kafka-topics --bootstrap-server kafka:29092 --list

    echo -e 'Creating kafka topics'
    kafka-topics --bootstrap-server kafka:29092 --create --if-not-exists --topic promethuesdata --replication-factor 1 --partitions 1

    echo -e 'Successfully created the following topics:'
    kafka-topics --bootstrap-server kafka:29092 --list
    "
```

docker-compose.yml --init-kafka

Il seguente microservizio viene utilizzato per inizializzare il topic usato per la comunicazione sopracitata in quanto in fase di testing abbiamo riscontrato un problema relativo alla creazione di quest'ultimo.

ETL data pipeline

Il microservizio ETL data pipeline si occupa, per ognuna delle 5 metriche esposte(availableMem, CpuUsage, cpuTemp, totProcesses, networkThroughput)di calcolare:

- Set di metadati con i relativi valori (autocorrelazione, stazionarietà e stagionalità).
- il valore di max, min, avg, dev_std della metriche per 1h,3h, 12h.
- Il valore di max, min, avg nei successivi 10 minuti per un set ristretto di metriche provenienti dallo SLA.

```
4 etl_data_pipeline:
5   build:
6     context: .
7     dockerfile: ./etl_data_pipeline/Dockerfile
8   command: python ./etl_data_pipeline/main.py
9   ports:
10    - "5000:5000"
11   volumes:
12    - ./etl_data_pipeline
13   networks:
14    - mynet
15
```

docker-compose.yml --init-kafka

Scraping dati prometheus

Il micro-servizio, per ogni metrica scelta, chiamerà la funzione qui di fianco per estrapolare gli opportuni metric data dal server Prometheus usando la funzione `get_metric_data` fornita dalla libreria `prometheus_api_client` che dato in ingresso gli opportuni parametri permette di effettuare la sopracitata azione.

```
25 def configureParams(name, label, start_time, end_time, chunk_size):
26     start = time.time()
27     print(start_time)
28     metric_data = prom.get_metric_range_data(
29         metric_name=name,
30         label_config=label,
31         start_time=start_time,
32         end_time=end_time,
33         chunk_size=chunk_size,
34     )
35     end = time.time()
36     return metric_data, (end - start)
37
```

Getting Prometheus data

Elaborazione dati

1. **Elaborazione Metadata:** per il calcolo di autocorrelazione, stazionarietà e stagionalità è stata utilizzata la libreria `statsmodel` e in particolare le funzioni di:
 1. `seasonal_decompose` -> stagionalità
 2. `Adfuller` -> stazionarietà
 3. `tsa.acf` -> autocorrelazione
2. **Elaborazione Metrics:** per ogni intervallo di tempo indagato (1h, 3h e 12h) vengono calcolati max, min, avg e std a partire dai valori ritornati dal server prometheus mostrando il tempo necessario per completare tale operazione.

```
for i in range(5):
    for x in start_time:
        label_config={'nodeName': nodeName[i], 'job': job[i]}
        #print(label_config)
        metric_data, diff_meta = configureParams(metric_name[i], label_config, parse_datetime(x), end_time, chunk_size)

        tempoExc[metric_name[i]] = {"time metrics": str(diff_meta) + " s"}
        print("Metric name : " + metric_name[i])
        print(tempoExc[metric_name[i]])

        metric_object_list = MetricsList(metric_data)
        metric_df = MetricRangeDataFrame(metric_data)

        key_dict = metric_name[i] + '-' + x
        time_execution_start = time.time()
        result2[key_dict] = calculate_values(metric_name[i], metric_df, x)
        time_end_execution = time.time()
        print("Tempo di esecuzione " + metric_name[i] + " : " + str(time_end_execution - time_execution_start) + " s")

print("Calcolo di metrics completato\n")
print("Invio al db in corso...\n")
```

3. **Elaborazione Prediction:** ricevuti i dati dallo SLA il micro-servizio si occupa di andare a ricavare max, min e avg delle metric prediction nei prossimi 10 minuti.
4. Tutti i dati esposti precedentemente vengono, a fine elaborazione, inviati tramite kafka al data storage

```
2023-01-30 19:29:43 {'time metadata': '161.05191278457642 s'}
2023-01-30 19:29:43 Tempo di esecuzione networkThroughput : 0.03037738800048828 s
2023-01-30 19:29:43 Calcolo dei metadati completato
2023-01-30 19:29:43
2023-01-30 19:29:43 Invio al db in corso...
2023-01-30 19:29:43
```

```

13
14 data_storage:
15   build:
16     context: .
17     dockerfile: ./data_storage/Dockerfile
18   command: python ./data_storage/main.py
19   ports:
20     - "5001:5000"
21   volumes:
22     - ./data_storage
23   depends_on:
24     - init-kafka
25

```

docker-compose.yml --data_storage

Data Storage

Il microservizio di data storage si occupa di gestire i dati strutturati ricevuti dall'ETL data pipeline, ottenuti sfruttando la connessione asincrona ottenuta tramite Kafka.

Connessione al database

Il microservizio, sfruttando la libreria mysql-connector-python, è stato collegato al database utilizzato come storage, settando opportunamente i parametri come mostrato in figura.

Viene inoltre definito l'oggetto "mycursor" che verrà sfruttato per interagire col database, in particolare per eseguire query.

```

data_storage > main.py > ...
23
24 mydb = mysql.connector.connect(
25     host="db",
26     user="user",
27     password="pswd",
28     database="DataStorage"
29 )
30 if mydb.is_connected():
31     print("connesso")
32
33 mycursor=mydb.cursor()
34

```

Configuration database connection

Comportamento microservizio

```

47 print("Aspetto un messaggio\n")
48
49 while True:
50
51     msg= c.poll(1.0)
52
53     if msg is None:
54         continue
55     if msg.error():
56         print("Consumer error: {}".format(msg.error()))
57         continue
58
59
60 > if str(msg.key())== "b'test'": ...
63
64 > elif str(msg.key()) == "b'file1'": ...
81
82 > elif str(msg.key()) == "b'file2'": ...
97
98 > elif str(msg.key()) == "b'file3'": ...
117

```

Gestione diversi tipi di dati in ingresso

Una volta instaurata la connessione tramite kafka, il microservizio resta in ascolto di un eventuale messaggio da parte del broker. Una volta ricevuto, esso ne distinguerà il contenuto tramite la key ad esso associato.

Nel particolare il data storage è in grado di gestire i seguenti casi:

- **Key b'file1:** dati associati alla tabella *metadata*, la quale contiene i dati riguardanti autocorrelazione, stazionarietà e stagionalità di una delle metriche sotto esame.
- **Key b'file2:** dati associati alla tabella *prediction*, la quale contiene i dati riguardanti massimo, minimo e valore medio dei valori predetti di una delle metriche sotto esame.
- **Key b'file3:** dati associate alla tabella *metrics*, la quale contiene i dati riguardanti massimo, minimo, valore medio e dev std di una delle metriche sotto esame.
- **Error:** in caso di errore il microservizio lo notificherà

Identificato il messaggio il data storage procede con la costruzione della query adatta al tipo di dati ricevuto in ingresso, di seguito l'esempio di come viene costruita la query usando l'oggetto *mycursor* per inserire i metadata ricevuti dall'ETL data pipeline:

```

58
59 > if str(msg.key()) == "b'test': ...
62
63 elif str(msg.key()) == "b'file1':
64     #tabella metadata metric_name/autocorrelazione/staz/seas
65
66     fileJson2 = json.loads(msg.value())
67     print(fileJson2)
68
69     sql = """INSERT INTO metadata(metric_name,autocorrelation,stationarity,seasonality) VALUES(%s,%s,%s,%s);"""
70
71     for key in fileJson2:
72         autocorrelation = str(fileJson2[key]["Autocorrelazione"])
73         stationarity = str(fileJson2[key]["Stazionarietà"])
74         seasonality = str(fileJson2[key]["Stagionalità"])
75         val=(key,autocorrelation,stationarity,seasonality)
76         print(val)
77         mycursor.execute(sql,val)
78         mydb.commit()
79         print("Ho caricato nel db")
80

```

Inserimento metadata

Data Retrieval

```

docker-compose.yml
27 networks:
28   - mynet
29
30
31 data_retrieval:
32   build:
33     context: .
34     dockerfile: ./data_retrieval/Dockerfile
35   command: python ./data_retrieval/main.py
36   ports:
37     - "5002:5002"
38   volumes:
39     - ./data_retrieval
40   depends_on:
41     - db
42   container_name: dataretrieval
43   networks:
44     - mynet
45

```

docker-compose.yml --data_retrieval

Il micro servizio di data retrieval permette di estrarre dati strutturati riguardanti le 5 metriche sotto esame attraverso delle GET API.

In particolare saranno disponibili per ogni metrica:

- QUERY dei metadata.

- QUERY dei valori max, min, avg,dev_std per le ultime 1,3,12 ore.
- QUERY dei valori predetti

```

17
18 @app.get('/')
19 > def home_page(): ...
22
23 @app.get('/metrics')
24 > def get_metrics(): ...
47
48 @app.get('/predict')
49 > def get_predict(): ...
68
69 @app.get('/metadata')
70 > def get_meta(): ...
91
92
93 if __name__ == "__main__":
94     print("\nMain data_retrieval")
95     get_metrics()
96     get_predict()
97     get_meta()
98     app.run(port = "5002",host="0.0.0.0",debug=False)

```

GET API

Questo servizio è stato implementato sotto forma di app Flask, Flask è un framework di Python per realizzare applicazioni web.

```
4 app = Flask(__name__)
```

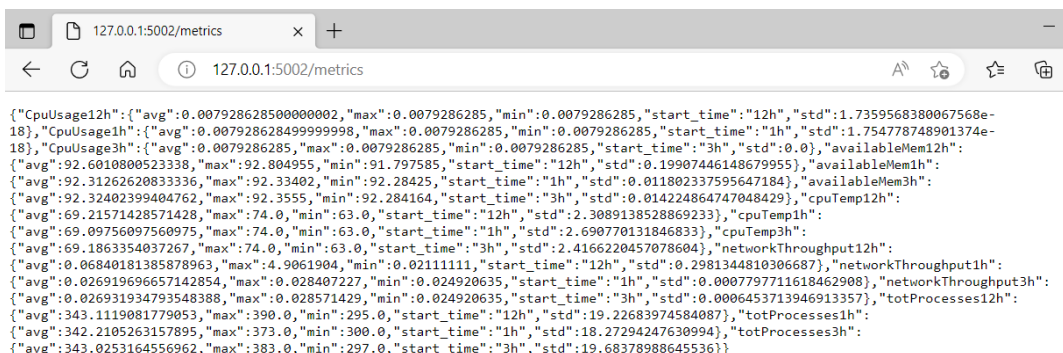
Flask framework

GET API

I dati ritornati dalle API provengono dal DB, anche in questo caso la connessione è stata effettuata tramite la libreria python mysql-connector-python.

Il micro-servizio offre le seguenti API:

1. **GET /metrics:** risponde con la rappresentazione Json della tabella *metrics* contenuta all' interno del DB, la quale conterrà i dati elaborate delle metriche sotto esame per 1h, 3h e 12h.



```

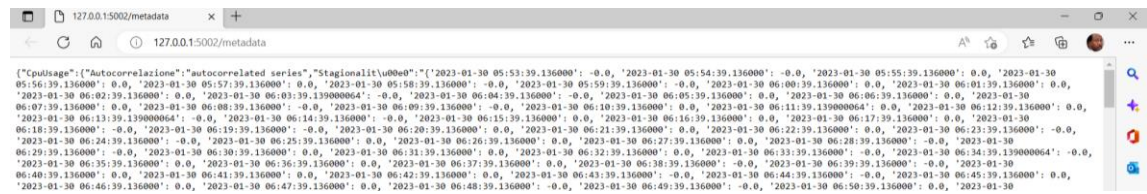
{"CpuUsage12h":{"avg":0.007928628500000002,"max":0.0079286285,"min":0.0079286285,"start_time":"12h","std":1.7359568380067568e-18},"CpuUsage1h":{"avg":0.007928628499999998,"max":0.0079286285,"min":0.0079286285,"start_time":"1h","std":1.754778748901374e-18},"CpuUsage3h":{"avg":0.0079286285,"max":0.0079286285,"min":0.0079286285,"start_time":"3h","std":0.0},"availableMem12h":{"avg":92.6010800523338,"max":92.804955,"min":91.797585,"start_time":"12h","std":0.19907446148679955},"availableMem1h":{"avg":92.31262620833336,"max":92.33402,"min":92.28425,"start_time":"1h","std":0.011802337595647184},"availableMem3h":{"avg":92.32402399404762,"max":92.3555,"min":92.284164,"start_time":"3h","std":0.014224864747048429},"cpuTemp12h":{"avg":69.21571428571428,"max":74.0,"min":63.0,"start_time":"12h","std":2.3089138528869233},"cpuTemp1h":{"avg":69.09756097560975,"max":74.0,"min":63.0,"start_time":"1h","std":2.690770131846833},"cpuTemp3h":{"avg":69.1863354037267,"max":74.0,"min":63.0,"start_time":"3h","std":2.4166220457078604},"networkThroughput12h":{"avg":0.06840181385878963,"max":4.9061904,"min":0.02111111,"start_time":"12h","std":0.2981344810306687},"networkThroughput1h":{"avg":0.026931934793548388,"max":0.028407227,"min":0.024920635,"start_time":"1h","std":0.0007797711618462908},"networkThroughput3h":{"avg":0.026931934793548388,"max":0.028407227,"min":0.024920635,"start_time":"3h","std":0.0006453713946913357},"totProcesses1h":{"avg":343.1119081779053,"max":390.0,"min":295.0,"start_time":"12h","std":19.22683974584087},"totProcesses12h":{"avg":342.2105263157895,"max":373.0,"min":300.0,"start_time":"1h","std":18.27294247630994},"totProcesses3h":{"avg":343.0253164556962,"max":383.0,"min":297.0,"start_time":"3h","std":19.68378988645536}}

```

GET API metrics

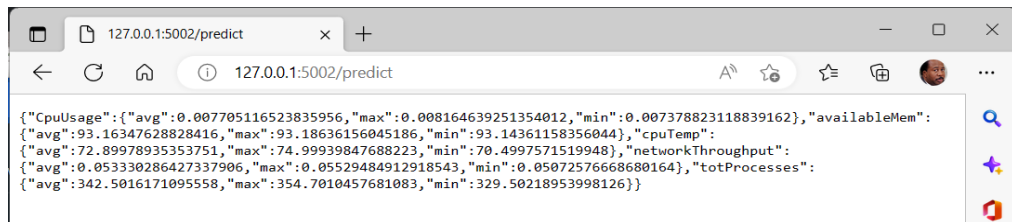
2. **GET /metadata:** risponde con la rappresentazione Json della tabella metadata contenuta all' interno del DB. Per via della dimensione assai estesa del campo value

legato alla stagionalità, il seguente campo è stato conservato nel db usando MEDIUMTEXT.



GET API metrics

3. **GET /predict:** risponde con la rappresentazione Json della tabella predict contenuta all'interno del DB.



GET API metrics

SLA Manager

È stato creato un micro servizio REST API per definire uno SLA caratterizzato da un set di 5 metriche e i relativi range di valori ammessi.

Esso restituisce, anche in questo caso come json, il numero di violazioni nelle ultime 1h, 3h e 12h e un indicazione sulle possibili violazioni nei prossimi 10 minuti. Inoltre l'inizializzazione del set da 5 metriche avviene tramite una POST sfruttando un API tester, a cui segue una successiva POST che invia all'ETL data pipeline i valori predetti su cui quest'ultimo andrà ad effettuare alcune elaborazioni.

```
64 sla_manager:
65   build:
66     context: .
67     dockerfile: ./sla_manager/Dockerfile
68   command: python ./sla_manager/main.py
69   ports:
70     - "5003:5003"
71   volumes:
72     - ./sla_manager
73   networks:
74     - mynet
75
```

docker-compose.yml --SLA Manager

TALEND API TESTER

Il set di metriche su cui verranno svolti i check sulle violazioni viene passato al microservizio tramite una POST API, che manderà in forma strutturata le 5 metriche e i relativi range di valori ammissibili.

Per fare ciò è stato utilizzato lo strumento di progettazione di API "Talend API Tester" che ha permesso di mandare il seguente JSON al microservizio.

The screenshot shows the Talend API Tester interface. At the top, there's a 'DRAFT' label and a 'Save as' button. The main configuration area includes a 'METHOD' dropdown set to 'POST', a 'SCHEME // HOST [":" PORT] [PATH ["/" QUERY]]' field containing 'http://127.0.0.1:5003/SLA', and a 'Send' button. Below this, there's a 'QUERY PARAMETERS' section. The 'HEADERS' section on the left has a 'Content-Type' header set to 'application/json'. The 'BODY' section on the right is set to 'Text' and contains a JSON payload:


```

1 {
2   "availableMem": {"min": 93.99379, "max": 94.030594},
3   "CpuUsage": {"min": 0.0079286285, "max": 0.0079286285},
4   "cpuTemp": {"min": 64.0, "max": 73.0},
5   "totProcesses": {"min": 301.0, "max": 377.0},
6   "networkThroughput": {"min": 0.019841272, "max": 0.021587303}
7 }
  
```

 At the bottom right, there are buttons for 'Top', 'Bottom', '2Request', 'Copy', and 'Download'.

Talend API Tester

Una volta ricevuto in POST un json di tale formato, lo SLA manager andrà ad elaborare le metriche, calcolando le violazioni rispetto ai valori passati e alle previsioni future, e si occuperà di inviare tramite una POST i risultati ottenuti.

Post API -> ETL data pipeline

Sfruttando il network "mynet" è stato possibile inviare i dati predetti ad un altro container all'interno della stessa rete utilizzando direttamente il container_name definite nel *docker-compose.yml*.

```

106
107 res=requests.post('http://etl_data_pipeline:5000/ETL',json=json1)
108 print(res.json)
109
  
```

POST API -> ETL data pipeline

```

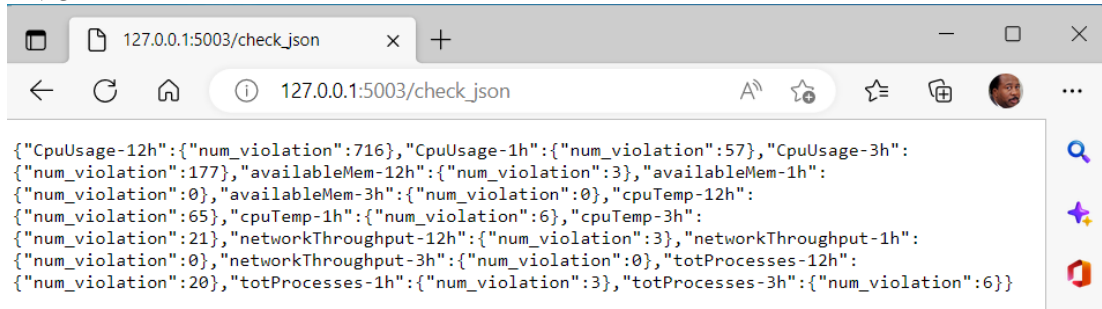
2023-01-30 19:46:06
2023-01-30 19:46:06 Invio al db in corso...
2023-01-30 19:46:06
2023-01-30 19:46:06 Invio completato
2023-01-30 19:46:06
2023-01-30 19:46:06 * Serving Flask app 'main'
2023-01-30 19:46:06 * Debug mode: off
2023-01-30 19:29:45 % Message delivered to prometheusdata [0] @ 0
2023-01-30 19:46:06 % Message delivered to prometheusdata [0] @ 1
2023-01-30 19:46:06 WARNING: This is a development server. Do not use it in a production
2023-01-30 19:46:06 ver instead.
2023-01-30 19:46:06 * Running on all addresses (0.0.0.0)
2023-01-30 19:46:06 * Running on http://127.0.0.1:5000
2023-01-30 19:46:06 * Running on http://172.30.0.3:5000
2023-01-30 19:46:06 Press CTRL+C to quit
2023-01-30 20:22:48 % Message delivered to prometheusdata [0] @ 2
2023-01-30 20:22:48 172.30.0.4 - - [30/Jan/2023 19:22:48] "POST /ETL HTTP/1.1" 200 -
  
```

Ricevuto il json dallo SLA tramite POST, l'ETL data pipeline elabora i dati strutturati e tramite Kafka li invierà al data storage.

GET API

Il micro-servizio offre le seguenti API:

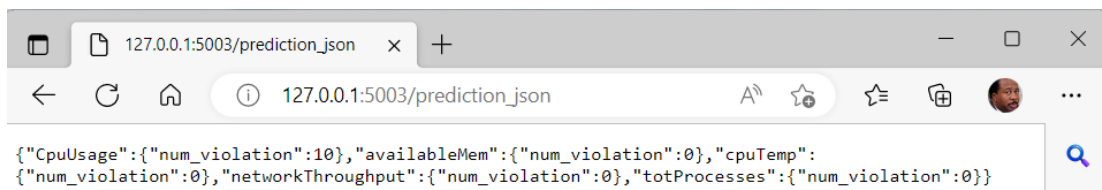
1. **GET /check_json:** il microservizio risponde con la rappresentazione Json del numero di violazioni calcolato sui dati provenienti dalle metriche rispettivamente per 1h, 3h e 12h.



```
{
  "CpuUsage-12h": {
    "num_violation": 716
  },
  "CpuUsage-1h": {
    "num_violation": 57
  },
  "CpuUsage-3h": {
    "num_violation": 177
  },
  "availableMem-12h": {
    "num_violation": 3
  },
  "availableMem-1h": {
    "num_violation": 0
  },
  "availableMem-3h": {
    "num_violation": 0
  },
  "cpuTemp-12h": {
    "num_violation": 65
  },
  "cpuTemp-1h": {
    "num_violation": 6
  },
  "cpuTemp-3h": {
    "num_violation": 21
  },
  "networkThroughput-12h": {
    "num_violation": 3
  },
  "networkThroughput-1h": {
    "num_violation": 0
  },
  "networkThroughput-3h": {
    "num_violation": 0
  },
  "totProcesses-12h": {
    "num_violation": 20
  },
  "totProcesses-1h": {
    "num_violation": 3
  },
  "totProcesses-3h": {
    "num_violation": 6
  }
}
```

GET API check_json

2. **GET /predict_json:** il microservizio risponde con la rappresentazione Json delle violazioni sui valori predetti sui dati provenienti dal server prometheus.



```
{
  "CpuUsage": {
    "num_violation": 10
  },
  "availableMem": {
    "num_violation": 0
  },
  "cpuTemp": {
    "num_violation": 0
  },
  "networkThroughput": {
    "num_violation": 0
  },
  "totProcesses": {
    "num_violation": 0
  }
}
```

GET API prediction_json