



ELABORATO DSBD

DISTRIBUTED SYSTEM AND BIG DATA

ANNO ACCADEMICO

2022/2023

PROPOSTO DA

ANDREA BATTAGLIA
ENRICA SPATARO

PANORAMICA DEL PROGETTO

OBIETTIVO

Il progetto di questo corso si pone come obiettivo quello di realizzare un sistema di monitoraggio di metriche.

La realizzazione di questo progetto si articola nella creazione di un insieme di microservizi per creare, ricercare, immagazzinare e prelevare informazioni circa le metriche esposte dall'exporter tramite un server Prometheus.

METODO

Dopo la creazione di un Docker container tramite docker-compose, si sono implementati i quattro microservizi richiesti.

La comunicazione tra questi ultimi avviene mediante il servizio di messaggistica Kafka.

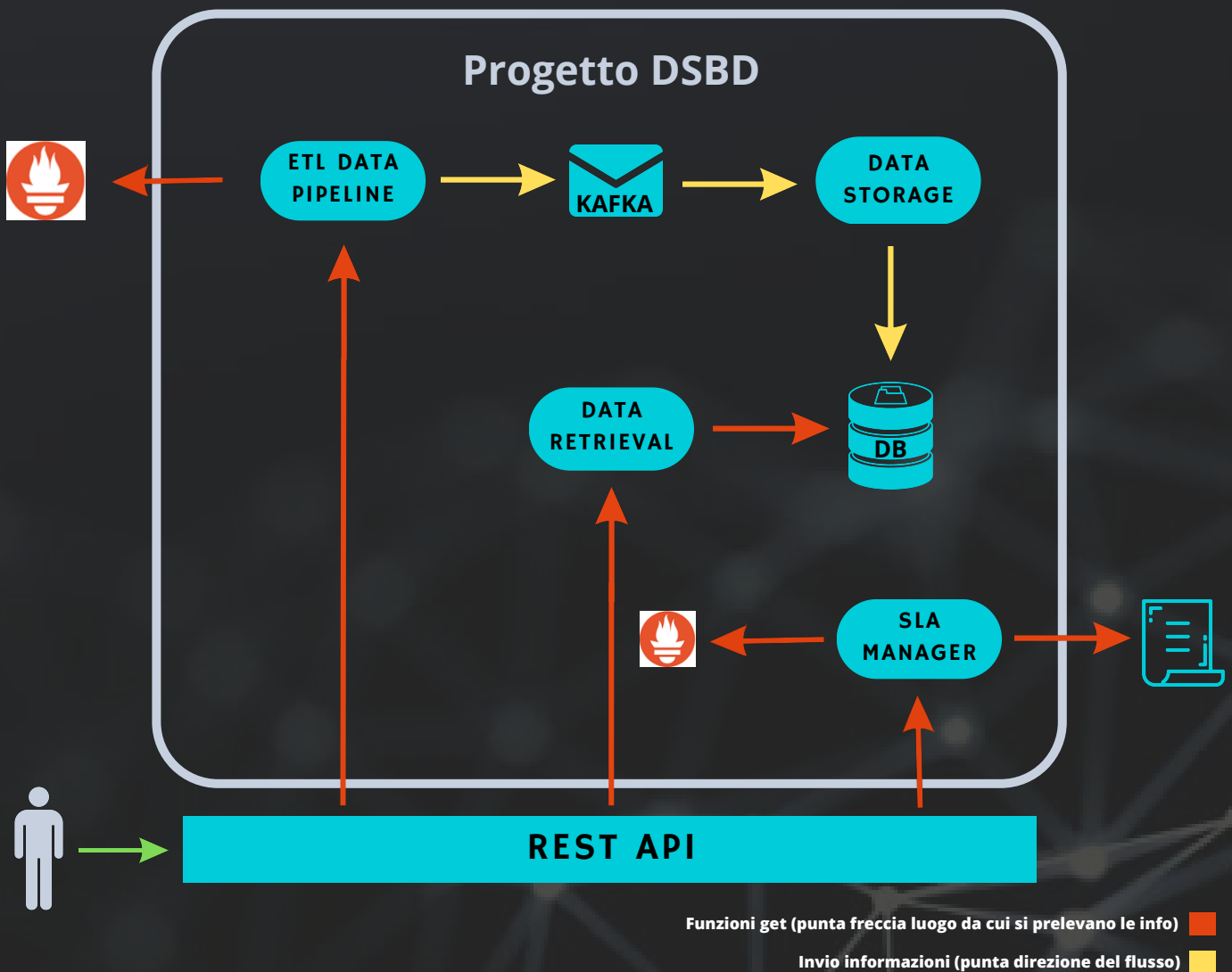
I dati creati e ricevuti vengono memorizzati in un database mysql.

RIEPILOGO

Dopo aver configurato un server Prometheus per fare scraping delle metriche, si è passati all'implementazione del sistema. Il sistema è composto da quattro microservizi:

- **Etl Data Pipeline**
- **Data Storage**
- **Data Retrieval**
- **SLA Manager**

DOCKER



All'interno di Docker è stato creato un container nominato "Progetto DSBD" che contiene vari microservizi.

L'ETL DATA PIPELINE preleva i dati da prometheus e, dopo averli elaborati, li invia tramite kafka al data storage che si occuperà di memorizzarli nel database.

I dati verranno prelevati dal data retrieval con delle query al database.

Lo SLA Manager, dopo aver prelevato i dati da un file json creato appositamente dallo script "sla_set.py", preleva da prometheus i dati necessari per le sue analisi.

Sono state implementate delle REST API, rispettivamente per il sistema di monitoraggio dell' ETL DATA PIPELINE e per fornire i dati del data retrieval e dello SLA Manager.

L'utente potrà accedere a queste REST API cliccando semplicemente nei link del file "index.html"

ABSTRACT APPLICAZIONE

L'idea sviluppata in questo progetto consiste in un insieme di microservizi che comunicano tra di loro all'interno di un container docker allo scopo di raccogliere, analizzare, immagazzinare e predire informazioni circa le metriche esposte dall'exporter tramite un server prometheus.

Il primo microservizio è l'ETL Data Pipeline che dopo aver raccolto, in seguito ad un opportuno filtraggio, le metriche da Prometheus, calcola i set di metadati, i valori significativi delle metriche in analisi (massimo, minimo, media e deviazione standard) e si occupa della predizione di alcuni valori per un set di metriche ristretto. Ogni operazione effettuata dall'ETL viene cronometrata: è stato implementato un sistema di monitoraggio interno che restituisce tramite REST API il tempo necessario per compiere le varie funzioni del microservizio.

Dopo aver elaborato i dati ed averli memorizzati in file json, essi vengono inviati tramite un messaggio Kafka con il topic "prometheusdata" al Data storage: questo secondo microservizio si occupa di prelevare le informazioni ricevute da Kafka ed immagazzinarle in un database di tipo relazionale.

Il terzo microservizio (Data Retrieval), a differenza del secondo, preleva i dati dal database in maniera strutturata e li fornisce al consumer tramite REST API.

L'ultimo microservizio è lo SLA Manager, che, implementato con interfaccia REST, permette di manipolare un set di 5 metriche con l'obiettivo di restituire l'eventuale numero di violazioni nelle ultime 1h, 3h e 12h; inoltre fornisce indicazioni su una possibile violazione nei successivi 10 minuti.

BUILD & DEPLOY

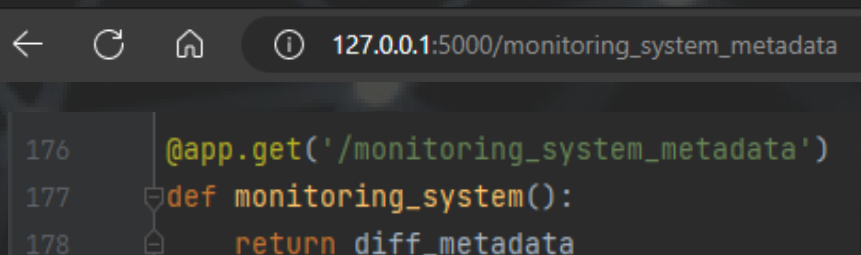
- Avviare il file "sla_set.py" (ProgettoDSBD > sla_manager > sla_set.py), in modo da generare il file json successivamente utilizzato nel microservizio 'sla_manager'
- Avviare il docker-compose presente nella cartella ProgettoDSBD
- Eseguire SUL BROWSER la pagina "index.html" per visualizzare le REST API ed il collegamento alla pagina del database

NOTE

1. Il microservizio init-kafka si stopperà subito dopo aver creato il topic;
2. I microservizi Data storage e Data retrieval presentano nel docker-compose il comando 'restart: always' perchè in questo modo riescono a collegarsi correttamente al database (comparirà un messaggio di errore e subito si riavviano in automatico);
3. I link contenuti nel file "index.html" vengono resi disponibili dopo la creazione dei vari microservizi (appena docker compose finisce di creare i suoi elementi); nel caso in cui non siano presenti elementi da mostrare la pagina restituirà un file Json vuoto "{}" (esempio: vengono fatte richieste alle funzioni del data retrieval prima che i dati vengano effettivamente immagazzinati nel database).
4. I link sono disponibili dal momento in cui nel log del microservizio appare una scritta del tipo: * Running on http://127.0.0.1:5002

NOTA BENE

Se non si volesse fare uso della pagina "index.html" è comunque possibile visualizzare i dati prodotti aggiungendo al path indicato da "Running on" il valore tra parentesi nell'app.get



The image shows a browser window with the address bar displaying "127.0.0.1:5000/monitoring_system_metadata". Below the browser, a code editor snippet shows the following Python code:

```
176 @app.get('/monitoring_system_metadata')
177 def monitoring_system():
178     return diff_metadata
```

Di seguito viene mostrato l'index.html

Welcome to Our DSDB Project

ETL Data Pipeline

Monitoring system to display the time taken for the execution of the metadata functions

[Monitoring System Metadata](#)

Monitoring system to display the time taken for the execution of the various functions

[Monitoring System Values](#)

Data Storage

click on the following link to connect to the database page

[Database datastorage](#)

Data retrieval

View the json file containing all metrics list

[Get all metrics](#)

View the json file containing the metadata autocorrelation, seasonality and stationarity values

[Get Metadata](#)

View the json file containing the metrics minimum, maximum, mean and standard deviation values

[Get metrics \(min,max,avg,std\)](#)

View the json file containing the 5 metrics predict values

[Get predictions](#)

SLA manager

Insert into console 5 metrics to monitoring

View the json file containing the 5 metrics sla/slo

[Get Sla with slo](#)

View the json file containing 5 metrics status (sla set)

[Get status SLA](#)

View the json file containing past violation for the 5 metrics (sla set)

[Get Past Violation](#)

View the json file containing possible future violation for the 5 metrics (sla set)

[Get Future Violation](#)

ETL DATA PIPELINE

Questo microservizio si occupa di calcolare e inoltrare dati.

Per ogni metrica esposta è stato:

- 1. Calcolato un set di metadati con i relativi valori**
- 2. Calcolato il valore massimo, minimo, medio e di deviazione standard per ogni metrica per 1h, 3h e 12h**
- 3. Predetto il valore massimo, minimo e medio per i successivi 10 minuti per un set ristretto di metriche**
- 4. Inoltrato in un topic Kafka "prometheusdata" un messaggio contenente i valori calcolati**

È inoltre stato creato un sistema di monitoraggio interno che rende visibile tramite REST il tempo necessario all'esecuzione delle varie funzionalità

CALCOLO SET DI METADATI

Dopo aver prelevato le metriche da analizzare attraverso una funzione di filtraggio che elimina le metriche con valori nulli e ne sceglie randomicamente un numero impostabile tramite il parametro 'metric_number' all'interno della funzione choose_metrics() (è stato testato fino ad un numero di 20 metriche), si è passati all'impostazione dei parametri per procedere con la query di Prometheus. I dati sono stati, per prima cosa, trasformati con la funzione MetricRangeDataFrame (operazione effettuata per tutti i dati prelevati da prometheus). Per il calcolo dei metadati sono stati prelevati campioni per almeno 24h; in seguito sono state invocate 3 funzioni per poter analizzare i valori ottenuti e ritornare valori utili per il calcolo dell'autocorrelazione, della stazionarietà e della stagionalità. Per verificare se una serie fosse o meno stazionaria è stato analizzato il P-value, verificando che quest'ultimo risulti inferiore a 0.05.

```
125 def autocorrelation(values):
126     lags = len(values)
127     result = 'non autocorrelated series'
128     result_autocorr = sm.tsa.acf(values, nlags=lags-1).tolist()
129     for lag in result_autocorr:
130         if lag <= 0.05:
131             result = 'autocorrelated series'
132
133     return result
134
```

```
105 def stationarity(values):
106
107     stationarityTest = adfuller(values, autolag='AIC')
108
109     if stationarityTest[1] <= 0.05:
110         result = 'stationary series'
111     else:
112         result = 'no stationary series'
113
114     return result
```

Per quanto riguarda l'autocorrelazione sono stati analizzati tutti i lags prelevati. Se almeno uno di questi risulta essere minore di 0.05 viene inserita una label che indica o meno l'autocorrelazione della serie.

Per la stagionalità sono stati prelevati e serializzati i valori ritornati dalla funzione seasonal_decompose.

```
117 def seasonal(values):
118     result_seasonal = seasonal_decompose(values, model='additive', period=5)
119     serializable_result = {str(k): v for k, v in result_seasonal.seasonal.to_dict().items()}
120
121     return serializable_result
122
```

CALCOLO VALORI DI MINIMO, MASSIMO, MEDIA E DEVIAZIONE STANDARD

Dopo aver selezionato le metriche, con un opportuno filtraggio che elimina valori nulli, si è calcolati il massimo, il minimo, la media e la deviazione standard di tutti i valori della singola metrica. In questa funzione il valore settato nel parametro start-time è stato fatto variare per 1h, 3h, 12h all'interno di un ciclo for.

```
96 # this function is used to calculate the max, min, avg, dev_std value of the metric for 1h, 3h, 12h;
97 def calculate_values(metric_name, metric_df, start_time):
98     file_json[str(metric_name + "," + str(start_time))] = {"max": metric_df['value'].max(),
99                                                             "min": metric_df['value'].min(),
100                                                             "avg": metric_df['value'].mean(),
101                                                             "std": metric_df['value'].std()}
102     return file_json
```

PREDIZIONE DI MINIMO, MASSIMO, MEDIA PER SUCCESSIVI 10 MINUTI

Dopo aver selezionato un set di 5 metriche e aver collezionato valori per 24h è stata implementata la funzione di predizione, utilizzando le funzioni:

- ExponentialSmoothing
- tsmodel.forecast

Sono stati predetti valori per i successivi 10 minuti.

Per ogni metrica è stato analizzato il trend per verificare se quest'ultimo fosse additivo o moltiplicativo ed è stato cercato il periodo adeguato per ottenere il residuo prossimo allo 0.

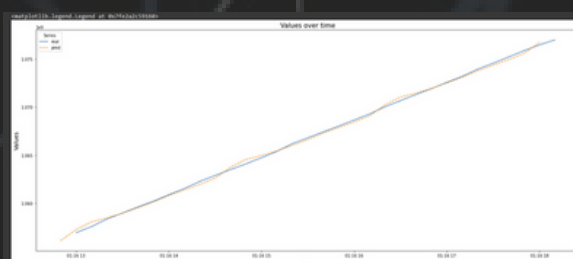
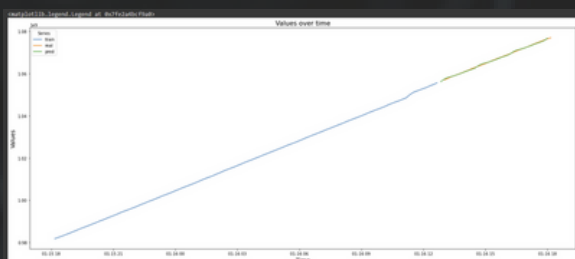
Inoltre è stato verificato che l'errore fosse inferiore alla deviazione standard tramite alcuni script in Pytorch per assicurarsi che i parametri invocati nella funzione del microservizio fossero settati correttamente.

Di seguito sono riportati due esempi.

ceph_bluefs_bytes_written_wal

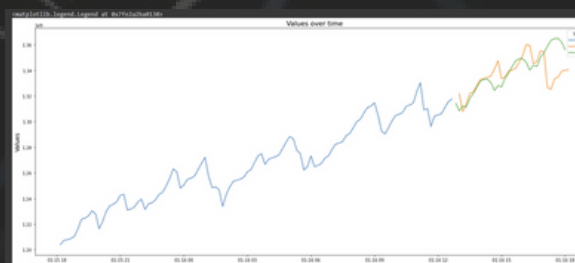
Questa metrica, come le altre 5 selezionate, presenta un trend additivo.

Come si può notare dai grafici, dopo aver allenato la predizione splittando in train data e test data i valori precedentemente collezionati (procedimento seguito per tutte e 5 le metriche selezionate), la predizione risulta alquanto accurata.



ceph_cluster_total_used_bytes

Come nel caso della precedente metrica questa predizione, seppur meno banale, risulta essere abbastanza accurata: in tutti e 5 i casi delle metriche analizzate l'errore (calcolato con la funzione "mean_absolute_error" e con la radice quadrata della funzione "mean_squared_error", della libreria sklearn.metrics) risulta essere sempre minore della deviazione standard (std)



INOLTRO DEL MESSAGGIO CONTENENTE I VALORI CALCOLATI IN UN TOPIC KAFKA "PROMETHEUSDATA"

Dopo aver raccolto i dati richiesti dai punti precedenti è stato creato un producer Kafka ed il relativo messaggio contenente un json con i valori precedentemente calcolati.

Viene inviato un messaggio per ogni tipologia di dati prodotti:

- metadati
- min, max, mean e dev_std
- valori predetti

```
69 # this function is used to set the parameters of the kafka producer and to create the message to be sent with the topic "prometheusdata"
70 def kafkaJsonProducer(fileJson, key):
71     broker = "kafka:29092"
72     topic = "promethuesdata"
73     conf = {'bootstrap.servers': broker}
74     p = Producer(**conf)
75
76     def delivery_callback(err, msg):
77         if err:
78             sys.stderr.write('%% Message failed delivery: %s\n' % err)
79         else:
80             sys.stderr.write('%% Message delivered to %s [%d] @ %d\n' %
81                             (msg.topic(), msg.partition(), msg.offset()))
82
83     try:
84         record_key = key
85         record_value = json.dumps(fileJson)
86         print("Producing record: {}{}\t{}".format(record_key, record_value))
87         p.produce(topic, key=record_key, value=record_value, callback=delivery_callback)
88
89     except BufferError:
90         sys.stderr.write('%% Local producer queue is full (%d messages awaiting delivery): try again\n' %
91                         len(p))
92     p.poll(0)
93     sys.stderr.write('%% Waiting for %d deliveries\n,' % len(p))
94     p.flush()
```

DATA STORAGE

Questo microservizio si occupa della creazione e dell'avviamento di un consumer group del topic "promethuesdata". Dopo aver prelevato i dati ricevuti dal flusso di messaggi, inserisce questi ultimi in un Database di tipo relazionale.

Dopo aver settato i parametri relativi al database (nel caso in cui quest'ultimo non dovesse esistere verrà creato automaticamente) e al consumer (avendo effettuato la sottoscrizione al topic), è stato avviato un ciclo infinito che resta in ascolto di ogni messaggio contenente il topic "prometheusdata":

```
22 while True:
23     msg = c.poll(1.0)
24
25     if msg is None:
26         continue
27
28     if msg.error():
29         print("Consumer error: {}".format(msg.error()))
30         continue
```

Ad ogni messaggio viene associata in fase di inoltro una chiave univoca che nel data storage permette di distinguere le varie tipologie di dati.

Dopo aver fatto il confronto tra il msg.key() e la relativa chiave viene effettuata una query sul database e vengono inseriti i dati.

Di seguito viene riportato un esempio di quanto appena scritto.

```
31 if str(msg.key()) == "b'etl#1'":
32     fileJson2 = json.loads(msg.value())
33     print(fileJson2)
34
35     sql = """INSERT INTO metadata (metric_name, autocorrelation, stationarity, seasonality) VALUES (%s,%s,%s,%s);"""
36
37     for key in fileJson2:
38         autocorrelation = str(fileJson2[key]["autocorrelazione"])
39         stationarity = str(fileJson2[key]["stazionarietà"])
40         seasonality = str(fileJson2[key]["stagionalità"])
41         val = (key, autocorrelation, stationarity, seasonality)
42         mycursor.execute(sql, val)
43         mydb.commit()
44         mycursor.execute("SELECT * FROM metadata;")
45
46     for x in mycursor:
47         print(x)
```

DATA RETRIEVAL

Questo microservizio, con interfaccia REST, permette di estrarre in modo strutturato le informazioni generate dall'applicazione ETL e contenute nel DB.

Sono rese disponibili:

- QUERY di tutte le metriche disponibili in Prometheus
- Per ogni metrica
 - QUERY dei metadati.
 - QUERY dei valori max, min, avg, dev_std per le ultime 1h, 3h e 12h.
 - QUERY dei valori predetti

Dopo aver settato i parametri relativi al database sono state implementate nuove funzioni che consistono in delle query SQL per prelevare i dati dal DB.

Ogni funzione ritorna un file json che viene visualizzato tramite la route dettata dall' app.get().

Nel file index.html basta cliccare sul link apposito come mostrato in seguito.

```
18 @app.get('/metrics')
19 def get_values_metrics():
20     sql = """SELECT * from metrics;"""
21     mycursor.execute(sql)
22     fileJson_metrics = {}
23     list_metrics = []
24     y = 0
25     for x in mycursor:
26         list_metrics.append(x)
27         max_metric = list_metrics[y][1]
28         min_metric = list_metrics[y][2]
29         avg_metric = list_metrics[y][3]
30         std_metric = list_metrics[y][4]
31         fileJson_metrics[list_metrics[y][0]] = {"max": max_metric,
32                                                 "min": min_metric,
33                                                 "avg": avg_metric,
34                                                 "std": std_metric}
35         y = y + 1
36     return fileJson_metrics
37
```

Data retrieval

View the json file containing all metrics list

[Get all metrics](#)

View the json file containing the metadata autocorrelation, seasonality and stationarity values

[Get Metadata](#)

View the json file containing the metrics minimum, maximum, mean and standard deviation values

[Get metrics \(min,max,avg,std\)](#)

View the json file containing the 5 metrics predict values

[Get predictions](#)

SLA MANAGER

Questo microservizio, implementato con interfaccia REST, permette di manipolare un set di 5 metriche con l'obiettivo di restituire l'eventuale numero di violazioni nelle ultime 1h, 3h e 12h.

Inoltre fornisce indicazioni su una possibile violazione nei successivi 10 minuti.

Sono resi disponibili:

- **CREATE/UPDATE** del SLA
- **QUERY** dello stato dell'SLA -
- **QUERY** del numero di violazioni nelle ultime 1h, 3h e 12h
- **QUERY** su possibili violazioni future dell'SLA

Per rendere automatico il tutto viene creato un file json, tramite lo script "sla_set.py", in cui vengono memorizzate le 5 metriche selezionate tramite la funzione hello().

Si può scegliere 5 tra 20 metriche selezionate, oppure si può utilizzare il set di default (contenente anch'esso 5 metriche).

```
6 # this function is used to generate the vector containing the metrics to be analyzed and monitored with the sla manager
7 def hello(default_metrics, data):
8     metrics_sla = {}
9     print("\n*** Hello from SLA MANAGER ***\n")
10    choose = input("\nDo you want to choose metrics or to use defaults ones?\nSelect 1 for choose, 0 for default")
11    if choose == '1':
12        print("\n Select your 5 metric by using metric codes")
13        for key in data:
14            print("Metric code: " + str(key) + " - " + data.get(key) + "\n")
15
16        k = 0
17        while len(metrics_sla) < 5:
18            metric = input("\ninsert metric: ")
19            if 0 <= int(metric) <= len(data):
20                if data[metric] not in metrics_sla:
21                    metrics_sla.append(data[metric])
22                    k = k + 1
23            else:
24                print("Cannot insert duplicate metrics; Please insert another metric \n\n")
25        else:
26            print("This metric code is not valid. Please retry \n\n")
27    else:
28        metrics_sla = default_metrics
29
30    return metrics_sla
31
```

Per ogni metrica viene chiesto se si vuole inserire manualmente il valore minimo ed il valore massimo da utilizzare in seguito per valutare le eventuali violazioni. Se non si vuole inserire, ai valori massimo e minimo verrà assegnato il valore false: nello SLA Manager e più precisamente nella funzione create_jsonSLA, nel caso in cui si dovesse riscontrare questo valore negli attributi min e max, verrà fatta una query a Prometheus con uno starttime di 30 minuti e, tra tutti i valori ritornati, vengono selezionati il minimo ed il massimo nonché impostati per valutare le violazioni per la suddetta metrica.

```
47 for key in range(0, len(metrics)):
48     x = input(
49         'Do you want to insert min and max for metric ' + metrics[key] + '? y/n\n')
50     if x == 'y':
51         min = input("Insert min: ")
52         max = input("Insert max: ")
53         fileJsonSLA[metrics[key]] = {'min': min, 'max': max}
54         print("\n", fileJsonSLA[metrics[key]])
55     elif x == 'n':
56         fileJsonSLA[metrics[key]] = {'min': False, 'max': False}
57     else:
58         print("\nWrong input -> please retry!\n")
59         key = key - 1
```

Lo script sopra descritto verrà eseguito prima del docker-compose: lo SLA Manager, dopo essere avviato da quest'ultimo, invocherà la funzione di seguito riportata.

```
30 # this function is used to generate the json file which will contain all the sla manager metrics with the relative slos
31 def create_jsonSLA(filejson, label_config, start_time_range, end_time, chunk_size):
32     fileJsonSLA = {}
33     for k in filejson:
34         print("\nProcessing data...")
35         print(filejson)
36         if filejson[k]["max"] == False or filejson[k]["min"] == False:
37             metric_data = setting_parameters(k, label_config, start_time_range, end_time, chunk_size)
38             metric_df = MetricRangeDataFrame(metric_data)
39             fileJsonSLA[k] = {'min': metric_df['value'].min(), 'max': metric_df['value'].max()}
40             print(fileJsonSLA)
41         else:
42             fileJsonSLA[k] = filejson[k]
43
44     return fileJsonSLA
```

Dopo aver creato il file json con il nome di ogni metrica da analizzare ed i corrispettivi valori di massimo e minimo, vengono invocate 3 funzioni:

- `past_violation()` - si occupa di verificare se nelle ultime 1h, 3h e 12h (dati prelevati da Prometheus) ci sono stati dei valori che hanno sfiorato il range (min - max) definito per quella determinata metrica;
- `future_violation()` - si occupa, dopo aver predetto i valori per i successivi 10 minuti, di verificare se nelle ultime 1h, 3h e 12h ci sono stati dei valori che hanno sfiorato il range (min - max) definito per quella determinata metrica;
- `statusSLA()` - si occupa di restituire per ogni metrica un valore che indica se è stata o meno commessa una violazione nella corrente metrica; nel caso in cui non sia stata commessa nessuna violazione verranno ritornati i valori di massimo, minimo e media per quella metrica.

Tutti i valori prodotti dallo SLA Manager possono essere visualizzati come in tutti i casi precedenti tramite i link presenti nella pagina "index.html"

LISTA API IMPLEMENTATE

ETL DATA PIPELINE

- **Monitoring system metadata**
- **Monitoring system values**

DATA RETRIEVAL

- **Get all metrics**
- **Get metadata**
- **Get metrics**
- **Get prediction**

SLA MANAGER

- **Get sla/slo**
- **Get status sla**
- **Get past violation**
- **Get future violation**