# Monitoraggio delle metriche di un nodo darwin

Relazione di Maria Roggio e Elio Vinciguerra

CT~19/01/2023

# Sommario


1. Manuale di avvio……..…3
   1.1. Avvio locale……3
      1.1.1. Istruzioni avvio database MySQL con XAMPP……..……3
      1.1.2. Istruzione avvio Kafka e Zookeeper…….....……...3
      1.1.3. Istruzioni avvio node exporter……4
      1.1.4. Istruzioni avvio prometheus…….……..……4
      1.1.5. Istruzioni avvio microservizio gRPCSLAManager……..……4
      1.1.6 Istruzioni avvio microservizio DataStorage……..4
      1.1.7 Istruzioni avvio microservizio gRPCDataRetrieval……..4
   1.2. Avviare il database MySQL con Docker-Compose……5
      1.2.1 Istruzioni avvio docker compose……..…….5
      1.2.2. Istruzioni avvio node exporter……..……4
      1.2.3. Istruzioni avvio prometheus…….……..……5
      1.2.4. Istruzioni avvio microservizio gRPCSLAManager……..……6
      1.2.5. Istruzioni avvio microservizio DataStorage……..6
      1.2.6. Istruzioni avvio microservizio gRPCDataRetrieval……...…..6

2. Introduzione……7
3. SLA Manager ed ETL Data Pipeline……...…9
   3.1. SLAManager……..9
   3.2. ETL Data Pipeline……...10
   3.3. TimeExecution……12
4. Data Storage……13
5. Data Retrieval……..……14

# Capitolo 1: Manuale di avvio

Nel seguente paragrafo vi saranno una serie di istruzioni per l'avvio del servizio.
Prima di avviare il servizio,assicurarsi di aver scaricato tutte le librerie contenute nel file requirements tramite il comando 'pip3 install -r requirements.txt' oppure 'pip install -r requirements.txt' (requirements.txt è contenuto nella cartella principale del progetto).

A seguire sono riportate le modalità di avvio se si possiedono XAMPP e Kafka (paragrafi 1.1 e 1.2), successivamente saranno illustrate le modalità di avvio tramite Docker-Compose (paragrafo 1.3):

***Nota***: Node_exporter e prometheus non vengono installati in automatico, bisogna installarli a parte. Ci riferiremo alle cartelle dove risiedono tali applicativi con "cartella node_exporter" e "cartella prometheus".

Nel caso in cui non si stia utilizzando la modalità di avvio con docker-compose mancheranno anche mysql e kafka.

## **1.1** Avvio locale:

### **1.1.1** Istruzioni avvio database MySQL con XAMPP

Avviare XAMPP e avviare MySQLDatabase e Apache Web Server, successivamente immettere le seguenti query:

1. *CREATE DATABASE **PythonDBTest**;*
2. *USE **PythonDBTest**;*
3. *CREATE TABLE **metricsStats** ( ID INT AUTO_INCREMENT, metric varchar(255), tempoInserimento timestamp, typeValue varchar(255), max DOUBLE, min DOUBLE, avg DOUBLE, devstd DOUBLE, PRIMARY KEY (ID));*
4. *CREATE TABLE **seasonability** ( ID INT AUTO_INCREMENT, metric varchar(255), tempoInserimento timestamp, tempo varchar(255), valore DOUBLE, PRIMARY KEY (ID));*
5. *CREATE TABLE **stationarity** ( ID INT AUTO_INCREMENT, metric varchar(255), tempoInserimento timestamp, ADFTest DOUBLE, pValue DOUBLE, nLagsUsed DOUBLE, nObs DOUBLE, CriticalValue1 DOUBLE, CriticalValue5 DOUBLE, CriticalValue10 DOUBLE, PRIMARY KEY (ID));*
6. *CREATE TABLE **autocorrelation** ( ID INT AUTO_INCREMENT, metric varchar(255), tempoInserimento timestamp, a DOUBLE, PRIMARY KEY (ID));*
7. *CREATE TABLE **predStats** ( ID INT AUTO_INCREMENT, metric varchar(255), tempoInserimento timestamp, max DOUBLE, min DOUBLE, avg DOUBLE, PRIMARY KEY (ID));*
8. *Controllare nel codice "dataStorage.py" (nella cartella principale del progetto) e "server.py" (nella cartella "gRPCDataRetrieval") i parametri di accesso.*

### 1.1.2 Istruzioni avvio Kafka e Zookeeper in locale

Sul prompt dei comandi entrare nella cartella Kafka, poi:

1. **Avviare Zookeeper**
   *./bin/zookeeper-server-start.sh config/zookeeper.properties*
2. **Avviare Kafka**
   *./bin/kafka-server-start.sh config/server.properties*
3. **Creare topic promethuesdata**
   *./bin/kafka-topics.sh --bootstrap-server localhost:29092 --topic promethuesdata --create*

### 1.1.3 Istruzioni avvio node exporter

Sul prompt dei comandi entrare nella cartella relativa a node exporter, poi:

1. **Avviare node exporter**
   *./node_exporter*

### 1.1.4 Istruzioni avvio prometheus

Sul prompt dei comandi entrare nella cartella prometheus, poi:

1. **Avviare prometheus**
   *./prometheus --config.file=prometheus.yml*

***NOTA:*** *Abbiamo modificato il file di configurazione di prometheus, è reperibile nella cartella principale del progetto sotto al nome prometheus.yml*

### 1.1.5 Istruzioni avvio microservizio gRPCSLAManager

Sul prompt dei comandi entrare nella cartella *gRPCSLAManager*, poi:

1. **Avviare proto3**
   *python3 -m grpc_tools.protoc -I. --python_out=. --pyi_out=. --grpc_python_out=. ./echo.proto*
2. **Avviare server**
   *python3 server.py*
3. **Avviare client**
   *python3 client.py*

### 1.1.6 Istruzioni avvio microservizio DataStorage

Sul prompt dei comandi entrare nella cartella principale del progetto, dove si trova il file Consumer.py, poi:

1. **Avvio Consumer**
   *python3 Consumer.py*

### 1.1.7 Istruzioni avvio microservizio gRPCDataRetrieval

Sul prompt dei comandi entrare nella cartella *gRPCDataRetrieval*, poi:

1. **Avviare proto3**

*python3 -m grpc_tools.protoc -I. --python_out=. --pyi_out=. --grpc_python_out=. ./echo.proto*

2. **Avviare server**
   *python3 server.py*
3. **Avviare client**
   *python3 client.py*

## **1.2** Avviare il database MySQL con Docker-Compose:

### **1.2.1** Istruzioni avvio docker compose

Sul prompt dei comandi entrare nella cartella principale del progetto, poi:

1. **Avviare docker compose**
   *docker-compose up*

Adesso si avvieranno mysql, zookeeper e kafka, rispettivamente sulle porte 3306, 2181 e 29092.

2. **Creare topic kafka**
   Il topic viene generato automaticamente dal Dockerfile.
3. ***Creazione database e tabelle:***
   Per la creazione del database e delle tabelle è possibile installare localmente un client MySQL e successivamente accedere con le seguenti credenziali:
   - MYSQL_ROOT_PASSWORD: prova
   - MYSQL_DATABASE: PythonDBTest
   - MYSQL_USER : progettoDSBD
   - MYSQL_PASSWORD: prova

   La procedura per la creazione del database e delle tabelle, a livello di query, è identica a come descritto nel paragrafo 1.1.1.

I passaggi a seguire sono i medesimi che si eseguono nella modalità in locale, li abbiamo riportati per comodità.

### **1.2.2** Istruzioni avvio node exporter

Sul prompt dei comandi entrare nella cartella relativa a node exporter, poi:

1. **Avviare node exporter**
   *./node_exporter*

### **1.2.3** Istruzioni avvio prometheus

Sul prompt dei comandi entrare nella cartella prometheus, poi:

1. **Avviare prometheus**
   *./prometheus --config.file=prometheus.yml*

***NOTA:*** *Abbiamo modificato il file di configurazione di prometheus, è reperibile nella cartella principale del progetto sotto al nome prometheus.yml*

### **1.2.4** Istruzioni avvio microservizio gRPCSLAManager

Sul prompt dei comandi entrare nella cartella *gRPCSLAManager*, poi:

1. **Avviare proto3**
   *python3 -m grpc_tools.protoc -I. --python_out=. --pyi_out=. --grpc_python_out=. ./echo.proto*

2. **Avviare server**
   *python3 server.py*
3. **Avviare client**
   *python3 client.py*

### 1.2.5 Istruzioni avvio microservizio DataStorage

Sul prompt dei comandi entrare nella cartella principale del progetto, dove si trova il file Consumer.py*,* poi:

1. **Avvio Consumer**
   *python3 Consumer.py*

### 1.2.6 Istruzioni avvio microservizio gRPCDataRetrieval

Sul prompt dei comandi entrare nella cartella *gRPCDataRetrieval*, poi:

1. **Avviare proto3**
   *python3 -m grpc_tools.protoc -I. --python_out=. --pyi_out=. --grpc_python_out=. ./echo.proto*
2. **Avviare server**
   *python3 server.py*
3. **Avviare client**
   *python3 client.py*

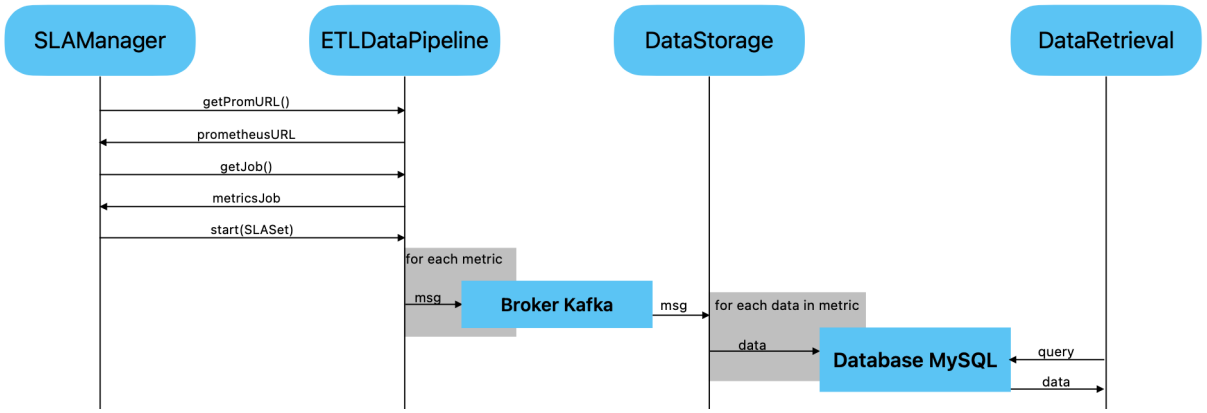
# Capitolo 2: Introduzione

Questo progetto verte sullo studio di metriche estratte da un nodo (dispositivo macOS) ed esposte tramite node exporter, per essere rese disponibili su server Prometheus, previa configurazione.
Le condizioni sufficienti all'avvio dell'attività, è che il node exporter e Prometheus siano già avviati per il tempo richiesto (es. 12h di attività corrispondono a 12h di metriche da analizzare).
Il lavoro che è stato svolto in questo progetto riguarda la creazione di microservizi per il calcolo, l'immagazzinamento e l'analisi di tutte le metriche rese disponibili da Prometheus o per un set di metriche scelte dall'utente finale.

In particolare viene data possibilità all'utente (client.py, il client gRPC del microservizio SLAManager) di selezionare 5 metriche da un elenco proposto e operare sui dati che mette a disposizione prometheus, non solo per capire quale metrica ha violato o violerà il range dello SLASet e di quanto, ma anche informazione in riferimento a massimi, minimi, media, deviazione standard, e metadati (autocorrelazione, stazionarietà e stagionalità) di tutte le metriche e previsione dei prossimi 10 minuti dei valori che potrebbero assumere le metriche dello SLASet.
Successivamente tali informazioni saranno mandate tramite un producer ad un topic kafka, il consumer leggerà tali informazioni e le memorizzerà in un database mysql.
L'ultimo microservizio parte proprio da tale presupposto, utilizzare il contenuto del database mysql per rendere disponibile ad un client gRPC i dati disponibili in riferimento ad una metrica selezionata.

Nel codice inviato insieme a questa documentazione abbiamo aggiunto una descrizione dei compiti di ogni funzione nei microservizi e incluso alcune note, tali commenti sono riportati dopo la definizione dell'intestazione della funzione in questione.

Segue un grafico riepilogativo delle comunicazioni tra i vari microservizi, (il meccanismo di funzionamento di ogni microservizio viene invece esplorato nei capitoli successivi):




Il microservizio SLAManager richiede il prometheus URL ad ETLDataPipeline, oltre al job, successivamente, in base alle scelte dell'utente, potrà richiamare la funzione start di ETLDataPipeline, che permette la creazione di minimo, massimo, media, deviazione standard, metadati e la predizione dei 10 minuti successivi sul valore di massimo, minimo e

media che assumeranno le metriche dello SLASet, successivamente ogni metrica, con le proprie informazioni, verrà inviata al broker kafka sul topic ‘prometheusdata’. Quando il consumer verrà avviato (microservizio DataStorage, verrà consumato ogni messaggio e i dati verranno salvati in un database mysql. Quando verrà avviato il microservizio DataRetrieval, ogni dato in relazione ad una metrica richiesta dall’utente verrà prelevato dal database e mostrato a schermo.

# Capitolo 3: SLAManager ed ETLDataPipeline

In questo capitolo verranno descritti due dei microservizi.
Il primo verte attorno alla realizzazione di una struttura gRPC client/server, che riceve input dal client (componente interattiva) e genera risposte dal server.
Il secondo verte attorno all'estrapolazione di dati (massimo, minimo, etc) da altre informazioni (metriche prometheus) o generazione di metadati a partire da informazioni già note.
In particolare tratteremo dei seguenti file contenuti in gRPCSLAManager: client.py, server.py, ETLDataPipeline.py, timeExecution.py e Producer.py

## *3.1: SLAManager*

Tramite il client gRPC (client.py) l'utente può scegliere un set di 5 metriche, SLA Set, in sostituzione di quello già proposto di default, ed associare a tale set dei parametri di valore massimo e minimo, che potranno essere utilizzati successivamente per calcolare il numero di violazioni, le violazioni stesse (nome della metrica e valore relativo alla violazione) oppure visualizzare se una data metrica di quello stesso SLASet nei 10 minuti successivi potrebbe eccedere il valore massimo del range o abbassarsi sotto al valore minimo. Tale client permette anche l'avvio di ETLDataPipeline, di cui verrà discusso in seguito.

Il client deve solo fare richieste al server gRPC (server.py), che elabora la richiesta e restituisce l'informazione in formato stringa, quindi, il server dev'essere il primo elemento da avviare tramite comando "python3 server.py".

***Scelte progettuali:*** *La prima scelta progettuale riguarda il  trattare uno SLASet di default, l'utente potrà visualizzarlo ed in seguito modificarlo, impostando valori di minimo e massimo in caso l'utente scegliesse di studiare eventuali violazioni.*
*Successivamente viene mostrato un menù in cui sarà possibile far scegliere all'utente  quale operazioni desidera effettuare.*
*Le operazioni sono le seguenti:*

- *Avviare ETLDataPipeline e mandare il messaggio dei valori risultati al topic Kafka;*
- *Richiedere una query sullo stato attuale dello SLASet, ovvero visualizzare le violazioni passate e le previsioni di violazioni;*
- *Richiedere il numero di violazioni dello SLASet nelle ultime 1h, 3h, 12h;*
- *Richiedere una previsione di violazione futura.*

*Verrà poi chiesto se l'utente desidera effettuare altre azioni.*

Nel caso in cui l'utente scelga la voce relativa ad ETLDataPipeline, potrebbe dover aspettare del tempo prima di ricevere la conferma che l'azione sia terminata, motivo per cui abbiamo implementato il server asincrono, esso riceve richieste, ma verranno implementate una dopo l'altra, non concorrenzialmente (nel modello sincrono avremmo potuto usare thread, ma abbiamo anche notato che i tempi di esecuzione delle singole attività crescevano

esponenzialmente in base al numero di richieste, in quanto fisicamente il server avrebbe dovuto computare in parallelo).

Altra scelta progettuale riguarda la funzione takeAllMetrics() che risiede nel server, essa richiama getAllMetrics(), ma c'è un commento immediatamente successivo che chiama la funzione getAllMetricsFromPrometheus(), la prima restituisce una lista di elementi, 18 nomi di metriche selezionate da noi, la seconda fa una richiesta a prometheus e filtra in base alle prime lettere del nome della metrica (abbiamo notato che node_exporter chiama tutte le sue metriche col prefisso "node_"), se si desidera utilizzare quest'ultima funzione al posto di getAllMetrics() basta decommentare la riga dove essa viene richiamata, in takeAllMetrics(). Questo filtraggio avviene poiché in ETLDataPipeline, abbiamo posto la label richiesta come parametro alla funzione get_metric_range_data() pari a "{'job': 'node_exporter'}", per utilizzare jobs diversi bisogna modificare manualmente quel parametro, dentro la funzione initialization() in ETLDataPipeline.py.
Come metriche di default sono state scelte le seguenti 18:
"node_power_supply_time_to_empty_seconds", "node_disk_written_bytes_total", "node_disk_written_sectors_total", "node_disk_read_bytes_total", "node_disk_read_sectors_total", "node_disk_write_errors_total", "node_disk_read_errors_total", "node_filesystem_avail_bytes", "node_filesystem_device_error", "node_filesystem_size_bytes", "node_memory_active_bytes", "node_memory_inactive_bytes", "node_memory_purgeable_bytes", "node_network_receive_packets_total", "node_memory_free_bytes", "node_network_transmit_multicast_total", "node_network_transmit_packets_total", "node_network_receive_multicast_total".
Per quanto riguarda, invece, lo SLASet di default, sono state selezionate le seguenti 5 metriche: "node_disk_written_bytes_total", "node_memory_free_bytes", "node_filesystem_avail_bytes", "node_memory_active_bytes", "node_network_receive_packets_total".

Un altro dettaglio che abbiamo scelto di implementare, sfruttando i meccanismi asincroni riguarda -lato client- la richiesta per la prima funzionalità, "Lancia ETLDataPipeline e manda i risultati al topic Kafka", si è optato per un'esecuzione asincrona del tutto, in quanto le informazioni ritornate dal server non servono successivamente (ritorna solo un messaggio di avvenuta conclusione della richiesta), quindi, l'utente, dopo aver effettuato tale richiesta, può effettuare altre operazioni senza attendere l'effettiva conclusione di ETLDataPipeline, che potrebbe impiegare del tempo.

## *3.2: ETLDataPipeline*

Tale microservizio esegue calcoli in base ai valori delle metriche che richiede a prometheus. Esso parte dalla funzione start() che verrà invocata dal server gRPC, e che richiede due parametri, la lista delle metriche e lo SLASet.

I valori delle metriche vengono prelevati -nella funzione initialization()- da prometheus tramite la funzione get_metric_range_data() e aggiunti in un dictionary la cui key è il nome stesso della metrica.

Successivamente, si procede con l'utilizzo di tali dati per la creazione di altri dati o l'estrazione di informazioni. La funzione parameters() si occupa di ricevere il dictionary con tutte le metriche al suo interno e calcolare, per ognuna di esse, il minimo, il massimo, il valore medio e la deviazione standard.
Viene poi, tramite la funzione predict, effettuata una predizione di 10 minuti dei valori che potrebbero assumere le metriche dello SLASet.

Come spiegato nel commento della funzione, ogni metrica viene convertita in un dataframe, successivamente viene fatto resampling a 5s e vengono passati i value alla funzione ExponentialSmoothing(), col metodo .interpolante() siamo in grado di interpolare (di default linearmente) i valori non validi (se non avessimo usato questa funzione ma .dropna() avrebbe potuto restituire un ValueWarning). Successivamente la predizione viene effettuata con il metodo forecast(), il cui parametro indica la quantità di campioni che deve prevedere, in particolare, round((10*60)/5) calcola 10 minuti in secondi e poi divide per 5sec, ovvero il tempo che intercorre tra un campione e l'altro, in modo da avere il numero di campioni necessari per una predizione di 10 minuti. Successivamente vengono calcolati massimo, minimo e media e viene inserito tutto in un dictionary.
Tutto ciò che riguarda, invece, i metadati (stazionarietà, autocorrelazione, stagionalità), è richiamato all'interno di valuesCalc(), se i valori che possiede la metrica non sono esclusivamente pari a zero, tali valori vengono passati alle relative funzioni di calcolo dei metadati, altrimenti viene restituito un dictionary vuoto, in quanto si è notato che se si fossero inviati comunque i dati, si potevano riscontrare problemi, ovvero il risultato dava valori non corretti o senza senso (NaN).

Le funzioni di calcolo dei metadati sono essenzialmente tre:

- stationarity(): tale funzione calcola la stazionarietà e inserisce alcuni parametri risultanti dentro un dictionary (tale scelta viene fatta poiché altrimenti non serializza con successo il messaggio che inviamo successivamente a kafka);
- autocorr(): restituisce una lista con all'interno i valori di autocorrelazione (la conversione a lista viene fatta per lo stesso motivo menzionato prima, se non venisse fatto, avremmo problemi a serializzare il messaggio da inviare a kafka);
- seasonal(): essa ritorna un dictionary coi relativi valori di seasonality (per la conversione a dictionary, stesso motivo dei precedenti), ma si è notato che più dati raccoglieva prometheus con node exporter attivo e più valori ottenevamo, riscontrando anche problemi con l'invio del messaggio kafka a causa del peso elevato. Una possibile soluzione potrebbe essere quella di modificare i file di configurazione di kafka per consentire l'invio di messaggi più pesanti, ma si è optato per un'altra soluzione, ridurre il numero di elementi della seasonality, infatti, i value in entrata sono registrati con una frequenza di 5 secondi, per tempi brevi, quando si hanno pochi dati, passarli tutti potrebbe anche essere una soluzione applicabile, ma col crescere dei dati disponibili diventa oneroso, sia in termini computazionali che a livello di spazio che occupa il messaggio, motivo per cui si è optato (quando possibile) per effettuare una scrematura iniziale, passare alla seasonal_decompose() solo un valore ogni 60, ovvero un valore ogni 5 minuti, come spiegato anche nel commento di tale funzione che è riportato nel codice.

In seguito verranno calcolati i dati temporali (come spiegato nel paragrafo 3.3: TimeExecution).

Dopo ciò verrà generato e mandato il messaggio al topic kafka, tramite la funzione generateAndSendMsgKakfa(), che riceve come parametri i valori calcolati (max, min, avg, devstd per le metriche ad 1h, 3h e 12h da prometheus, i metadati e le predizioni, oltre alla lista contenente il  nome delle metriche). Infine verrà definito un dictionary (includendo o meno la previsione, a seconda della sua presenza o meno) per inviare singolarmente ogni metrica al producer, tramite la funzione sendKafka() contenuta in Producer.py.

La funzione sendKafka() si occupa di ricevere il singolo messaggio e inoltrarlo al topic ‘prometheusdata’ in formato stringa, successivamente viene visualizzato un messaggio di successo o di errore tramite la funzione acked().

## *3.3: TimeExecution*

Per avere indicazione temporali su quanto tempo impiega ogni funzione ad eseguire i calcoli richiesti, si è fatto uso dei timestamp, che sono successivamente passati come parametri alla funzione sendTimeExe(), che scrive in un log (a livello informativo) tutte le informazioni. Tale funzione risiede nel file timeExecution.py. Tutti i log verranno salvati in monitoring.log (se non presente verrà creato il file).

# Capitolo 4: DataStorage

In questo capitolo verrà descritto il microservizio relativo al data storage.
Tale microservizio consente la ricezione di messaggi kafka e ne permette l'immagazzinamento in un database mysql.
In particolare tratteremo dei seguenti file contenuti nella cartella principale del progetto: Consumer.py, dataStorage.py.

Il relativo microservizio citato ad inizio paragrafo verte la memorizzazione delle metriche in arrivo al topic kafka all'interno di un database mysql.
Il Consumer si sottoscrive al topic prometheusdata, tramite la funzione subscribe(['prometheus']), in seguito prendiamo i dati contenuti al suo interno con la funzione di poll, controlliamo se vi siano errori, nel caso non ve ne siano si convertono i dati, che al momento sono in formato stringa, in un dictionary, infine viene chiamata la funzione saveData(), contenuto nello script dataStorage.py, passandogli i dati appena convertiti.

Il dataStorage, contiene tre funzioni, la prima di queste è la funzione per la connessione col database, in cui inseriamo i dati di configurazione e controlliamo se vi sono errori durante il collegamento, in tal caso si interrompe l'esecuzione del programma.
La seconda funzione è insertDb(), che, dato l'handler del database e una query, esegue quest'ultima.
Tale funzione è stata immaginata per l'inserimento di informazioni nel database mysql.
La terza funzione è il saveData, già citata nel Consumer, il cui compito è quello di scorrere tutta la struttura mandatale in precedentemente (come spiegato in precedenza, arriva una sola metrica per messaggio, ma se si volesse implementare un modo per mandare più metriche per messaggio, non cambierebbe l'implementazione di tale script) e richiamare la funzione insertDb(), in cui saranno passati il database e la query(nel formato "INSERT INTO tabella(campi) VALUES (valori)".
Questa funzione è quella che collega le precedenti due e viene chiamata dal consumer kafka. Per quanto riguarda i valori dei metadati e della predizione, che potrebbero non essere presenti (il primo, poiché se tutti i valori sono pari a zero, i risultati dei metadati risulterebbero inconsistenti, il secondo poiché la predizione viene effettuata solo sullo SLASet), viene effettuato un controllo, solo se esiste la chiave relativa, viene richiamata la funzione di inserimento con la relativa query.
Dopo l'inserimento di tutti i dati delle metriche, viene chiusa la connessione col database.

# Capitolo 5: Data Retrieval

In questo capitolo verrà descritto il microservizio relativo al data retrieval.
Tale microservizio verte attorno alla realizzazione di una struttura gRPC client/server, che permette la ricezione input dal client (componente interattiva) e genera risposte dal server, tali risposte riguardano le informazioni memorizzate nel database mysql citato nel capitolo 4.
In particolare tratteremo i seguenti file contenuti nella cartella gRPCDataRetrieval: client.py, server.py.

Tale microservizio è composta da una parte client (client.py) ed una server (server.py), la parte client è la componente interattiva, che riceve input dall'utente e visualizza a schermo le informazioni ricevute dal server (previa richiesta). In particolare all'interno dello script 'client.py' vi sono due funzioni, la prima (viewResp()) permette la visualizzazione delle informazioni ricevute dal server, che come parametri accetta la risposta ed un 'case', che indica il tipo di dato (metadato, max-min-avg-devstd e predizione). La seconda (run()) viene invocata all'avvio del client e permette di inviare richieste al server.
La componente server è formata da più funzioni, la prima (connDb()) permette la connessione al database, se esso non risulterà raggiungibile verrà restituito un errore.
La seconda funzione (takeData()) preleva dati dal database e li inserisce in liste.
All'interno della classe EchoService vi risiedono le funzioni che possono essere invocate dal client. tra queste:

- takeAllMetrics(): preleva tutti i nomi di metriche disponibili in metricsStats;
- takeMetadata(): prende tutti i metadati di una metrica inserita in input dall'utente, nel client, e che il server deve ricevere (se presenti);
- takeValues(): prende i valori di massimo, minimo, media e deviazione standard di una metrica inserita in input dall'utente, nel client, e che il server deve ricevere;
- takePred(): prende i valori di massimo, minimo, media della predizione dei valori assunti da una metrica inserita in input dall'utente, nel client, e che il server deve ricevere (nota: la predizione viene effettuata in riferimento ai 10 minuti successivi al calcolo della stessa).

***Scelte progettuali:*** *Nel caso ci siano diversi dati nel database, l'utente, potrebbe dover aspettare del tempo prima di ricevere la conferma che l'azione sia terminata, motivo per cui abbiamo implementato il server asincrono, esso riceve richieste, ma verranno implementate una dopo l'altra, non concorrenzialmente (nel modello sincrono avremmo potuto usare thread, ma abbiamo anche notato che i tempi di esecuzione delle singole attività crescevano esponenzialmente in base al numero di richieste, in quanto fisicamente il server avrebbe dovuto computare in parallelo).*