SBDIO I4.0
Big Data
for Industry

O5.3

Progetto e realizzazione di un servizio di gestione business oriented in grado di analizzare i dati ottenuti dalle macchine tramite la piattaforma Teorema, dal sistema informativo aziendale (CRM, ERP, ecc.) e da open data / social / crowdsourcing

Code	05.3
Date	
Type	Confidential
Participants	UNIFE
Authors	Mauro Tortonesi (UNIFE), Marco Govoni (UNIFE), Federico Frigo (UNIFE)
Corresponding Authors	Mauro Tortonesi

POR FESR EMILIA-ROMAGNA 2014/2020
FSC Fondo per lo Sviluppo e la Coesione
UNIONE EUROPEA Fondo europeo di sviluppo regionale
Regione Emilia-Romagna

SBDIO I4.0
Big Data
for Industry

Sommario

1. Architettura 1

2. Use case CIRI-MAM 4

2.1 Descrizione sorgente dati 4

2.2 Descrizione dati vibrazionali 6

2.3 Evoluzione della piattaforma 6

3. Implementazione 8

3.1. Gateway di raccolta dati 8

3.1.1 Gateway versione 2 11

3.2. Kafka e Kafka-Connect 12

3.4. Matlab 14

3.4.1 Codice 14

3.5. Nuove visualizzazioni su Kibana 19

4. Deploy, esecuzione e migrazione dei servizi 20

4.1. Deploy dei servizi su CIRI-ICT 20

4.2. Migrazione del cluster su T3Lab 21

1. Architettura

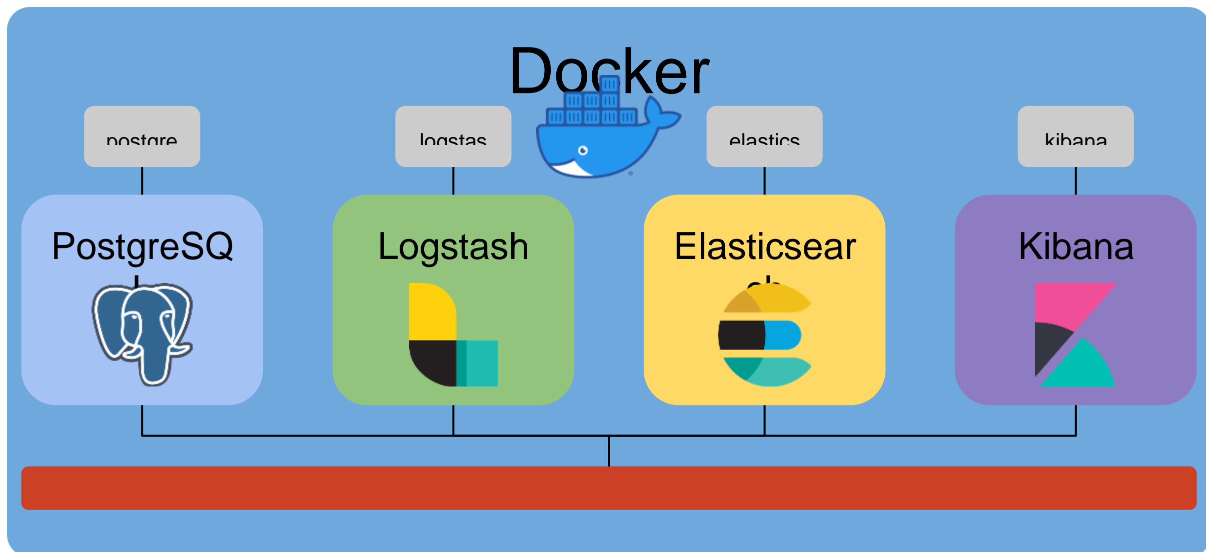


Fig. 1. Architettura della piattaforma su Docker

Come descritto nell'ultimo deliverable O5.2, la piattaforma per servizi OUT viene distribuita in fase di deployment sfruttando i **container Docker** (Figura 1). Questa scelta è motivata dal fatto di velocizzare lo sviluppo ed il testing dei vari componenti sfruttando i benefici dei container Docker. In questo scenario, è stato impiegato lo strumento **Docker Compose** per la definizione ed esecuzione di applicazioni multi-container. Docker Compose si occupa di automatizzare il download e il setup dei vari componenti, consentendo di eseguire e/o terminare in maniera molto semplice l'intera architettura.

La struttura presentata in Figura 1 è stata impiegata nel caso studio Carpigiani per l'abilitazione di diversi servizi (PostgreSQL come sorgente dati e stack ELK per storicizzazione ed analytics) oltre al network per la loro comunicazione (elk_net). Nel corso di questo documento vedremo com'è stato affrontato un secondo scenario applicato allo use case proposto dal CIRI-MAM nell'ambito dell'analisi di dati vibrazionali.

2. Use case CIRI-MAM

Nel corso dell'ultima fase di sviluppo prevista dal progetto è stato affrontato il caso studio proposto dal CIRI-MAM per l'impiego dei servizi OUT anche nell'ambito delle analisi di dati vibrazionali. Lo use case presentato in questo documento ha visto la collaborazione di Unife con il CIRI-MAM per quanto riguarda tutto ciò che concerne le sorgenti di dati vibrazionali, la loro storicizzazione e le analytics richieste da esperti di dominio. Per quanto riguarda invece la distribuzione della piattaforma e dei servizi abilitanti si è lavorato a stretto contatto con CIRI-ICT e T3Lab che hanno fornito tutte le risorse cloud necessarie per la loro esecuzione.

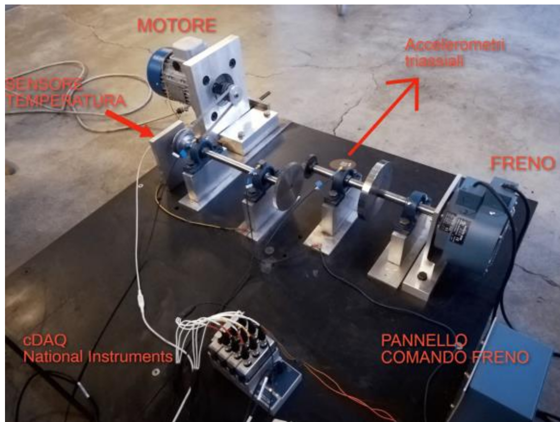
Entrando nel dettaglio del caso studio, il lavoro proposto da CIRI-MAM riguarda la realizzazione di un prototipo per il calcolo della stima del tempo di vita utile (RUL) nel caso di componenti meccanici. Attualmente, le fasi di acquisizione dati e calcolo dei parametri RUL sono operazioni fortemente slegate tra loro, passando da una prima fase di acquisizione per poi applicare in maniera completamente offline le analisi sui dati raccolti. Tali analisi, in generale, comportano una prima fase di "addestramento" del modello, segue una seconda fase dove il modello è pronto per essere applicato ai dati acquisiti, restituendo mano a mano le previsioni del tempo di vita utile RUL.

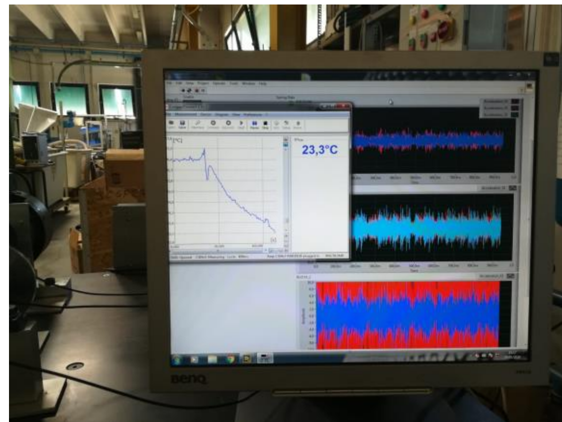
Nello scenario del CIRI-MAM, la prima fase di acquisizione dati è effettuata attraverso strumentazione apposita (es. accelerometri) e software apposito come ad esempio LabVIEW; le fasi invece di analisi dati sono tipicamente effettuate in MATLAB dopo aver importato in maniera completamente offline i dati desiderati. L'obiettivo proposto da CIRI-MAM comprende quindi un sistema innovativo per velocizzare le analisi RUL sfruttando la piattaforma attualmente impiegata nei servizi OUT del progetto SBDIO, nell'ottica di stabilire il più velocemente possibile la vita residua del componente meccanico sotto osservazione.

2.1 Descrizione sorgente dati

Il laboratorio meccanico del CIRI-MAM prevede un banco prova (Fig. 2) nel quale è installato un riduttore meccanico posto sotto stress e sollecitazioni di diversa natura. Tramite strumentazioni hardware apposite, come un Compact DAQ NI ed accelerometri industriali, è possibile acquisire in tempo reale i dati delle vibrazioni e monitorare gli andamenti dei parametri interessati.

Per far questo gli esperti di dominio impiegano attualmente il software LabView che consente loro di visualizzare e tenere traccia dei dati delle varie prove, tramite esportazione su file in formato LVM.





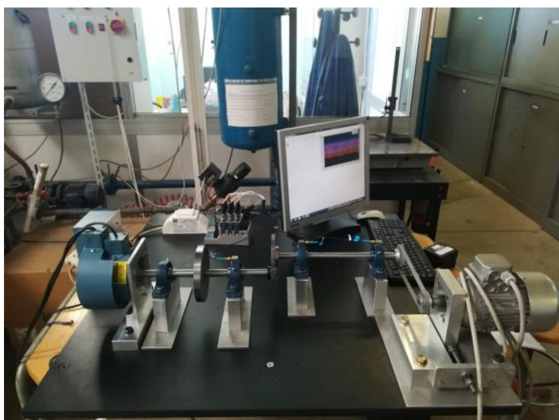


Fig. 2 Banco prova

2.2 Descrizione dati vibrazionali

Durante la fase di acquisizione dati vengono prodotti da LabVIEW una serie di file LVM contenenti i dati misurati. È possibile definire la dimensione massima di ogni file prodotto, come ad esempio 50 o 100MB.

In particolare, i dati vibrazionali sono strutturati come mostrato in Fig. 3.

È presente un header che contiene i metadati, come ad esempio riferimenti temporali, parametri tecnici della prova e frequenze di campionamento. Segue poi un payload strutturato a colonne, per ognuna delle quali vengono riportati tutti i campioni acquisiti.

Headers con metadati (data, ora, formati, …)									
X_Value	Acceleration_X1	Acceleration_Y1	Acceleration_Z1	Acceleration_X2	Acceleration_Y2	Acceleration_Z2	Acceleration_X3	Acceleration_Y3	Acceleration_Z3
4559,000000	-0,364879	0,537611	-0,612346	0,260307	0,937326	-0,105131	-1,001196	-1,661973	0,329297
…	…	…	…	…	…	…	…	…	…
4559,999922	1,329841	-0,364367	0,265228	0,969892	-0,480896	0,036098	1,606906	-3,075043	0,460747

Fig. 3 Struttura dati vibrazionali

2.3 Evoluzione della piattaforma

In Fig. 4 è illustrata la pipeline proposta per questo caso studio, che estende la pipeline già proposta per i servizi OUT del progetto. Nel lato sinistro troviamo la sorgente dati LabVIEW ed un gateway per la raccolta dati, mentre nella parte a destra è presente lo stack che si occupa della condivisione, storicizzazione ed analisi. Lo stack è stato inizialmente testato nel cloud di CIRI-ICT e poi è stato migrato nel cloud di T3Lab, a parte la macchina contenente Matlab che è rimasta in CIRI-ICT per una questione di licenze.

Le due sezioni, come evidenziato in figura, sono anche fisicamente separate. La sorgente dati è collocata nel laboratorio meccanico del CIRI-MAM, dove è presente una macchina Windows che esegue sia LabVIEW che un gateway di raccolta dati. Lo stack di analisi invece è eseguito grazie a risorse cloud messe a disposizione dal CIRI-ICT e da T3Lab nella propria infrastruttura cloud. Anche in questo scenario il deployment dello stack è effettuato attraverso container Docker su infrastruttura Kubernetes orientata ai microservizi.

L'obiettivo del gateway è di intercettare tutti i dati utili prodotti da LabVIEW e di inoltrare le informazioni contenute verso lo stack remoto di UniBO o di T3Lab, attraverso una connessione VPN che garantisce la sicurezza dei trasferimenti. Nel cluster remoto troviamo poi un broker Kafka responsabile di immagazzinare il più velocemente possibile i dati, per renderli disponibili ai servizi a valle interessati a processarli. Troviamo poi un componente Kafka Connect configurato per l'inoltro di tutti i dati, sia grezzi che elaborati (come vedremo in seguito) verso la base dati Elasticsearch. Tramite Kibana è quindi possibile realizzare dashboard di diversa tipologia in base alle analytics richieste.

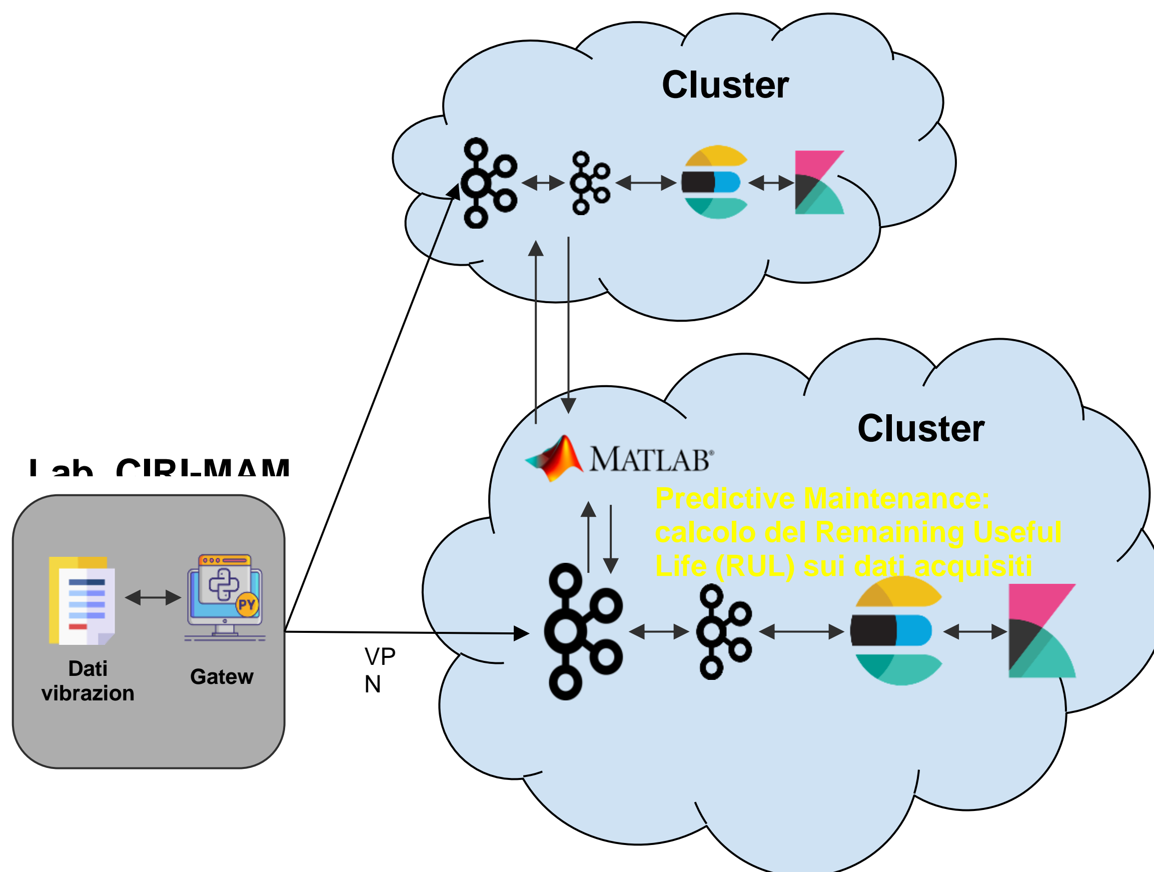


Fig. 4 Evoluzione dell’architettura servizi OUT

3. Implementazione

3.1. Gateway di raccolta dati

Si avvia con un doppio click sulla macchina Windows che ospita il simulatore LabView. Richiede una cartella locale da monitorare, che sarà la cartella di destinazione per i dati vibrazionali acquisiti tramite LabView. All’arrivo dei nuovi file, il Gateway estrae le informazioni necessarie e si occupa di trasferire i dati nel cluster remoto fornito da UniBO/T3Lab. Dai test effettuati, il Gateway risulta essere 5/6 volte più lento a consumare i dati rispetto alla velocità che impiega LabView a generarli; è comunque possibile ottimizzare questo comportamento per migliorare le performance del Gateway.

```
from watchdog.events import FileSystemEventHandler, FileModifiedEvent
from watchdog.observers import Observer
import glob
import json
from datetime import datetime, timedelta
from uuid import uuid4
from os import path

import lvm_read
from kafka import KafkaProducer
from time import perf_counter

to_be_processed_dir = '.'


def process_file(filename):
    print(f"Processing file {filename}\n")

    # read <filename>.lvm
    lvm = lvm_read.read(filename, read_from_pickle=False, dump_file=False)

    # file infos
    # print(lvm.keys())
    # print(lvm['Segments'])
    # print(lvm)

    # delete last 2 cols (useless)
    # data = np.delete(lvm[0]['data'], 11, 1)
    # data = np.delete(data, 10, 1)
    data = lvm[0]['data']

    # batch and tcp buffer sizes for Kafka
    batch_size = int(len(data) / 10)
    send_buffer_bytes = int(batch_size / 10)

    # kafka producer setup
    producer = KafkaProducer(bootstrap_servers=["kafkaot:9092"], value_serializer=lambda x:
json.dumps(
        x).encode('utf-8'), batch_size=batch_size, send_buffer_bytes=send_buffer_bytes)

    # generation of timestamp for keys
    dt = datetime.strptime(
        f"{lvm['Date']} {lvm['Time'].split(',').pop(0)}", "%Y/%m/%d %H:%M:%S")

    # print(f'Sending 0/{data.shape[0] * 3} message(s) to Kafka', end="\r")

    start_time = perf_counter()
```

SBDIO I4.0
Big Data
for Industry

```
    id = str(uuid4())

    # for each row, group each sensor values (x, y, z) and send to a Kafka topic (accelerometer)
    for row in data:
        dt_acc = (
            dt + timedelta(microseconds=row[0] * pow(10, 6))).isoformat(timespec="microseconds")
        accelerometer = {'id': id, 'timestamp': dt_acc, 'x1': row[1], 'y1': row[2], 'z1': row[3],
                         'x2': row[4], 'y2': row[5], 'z2': row[6], 'x3': row[7], 'y3': row[8], 'z3': row[9]}
        producer.send('accelerometer_raw', value=accelerometer,
                      key=f'{dt_acc}'.encode())
        # print(f'Sending {index * 3}/{data.shape[0] * 3} message(s) to Kafka', end="\r")

    # print(f'Sending {data.shape[0] * 3}/{data.shape[0] * 3} message(s) to Kafka\n')
    execution_time = perf_counter() - start_time

    producer.close()

    print(f"Processing of {filename} completed in {execution_time} seconds\n")
    print("\nWaiting for new files...")


class HandleFiles(FileSystemEventHandler):
    def on_modified(_, event: FileModifiedEvent):
        if not event.is_directory and event.src_path.split(".")[1] == "lvm":
            process_file(event.src_path)


if __name__ == "__main__":
    while True:
        to_be_processed_dir = input("Enter directory to track\n")

        if path.isdir(to_be_processed_dir):
            break
        else:
            print(f"{to_be_processed_dir} is not a valid path.")

    observer = Observer()
    observer.schedule(HandleFiles(), to_be_processed_dir, recursive=False)
    observer.start()

    filenames = glob.glob(f"{to_be_processed_dir}/*.lvm")

    if len(filenames) > 0:
        print("Processing files already present in target directory\n")
        for filename in filenames:
            process_file(filename)
    else:
```

POR FESR EMILIA-ROMAGNA 2014/2020
FSC Fondo per lo Sviluppo e la Coesione
UNIONE EUROPEA Fondo europeo di sviluppo regionale
RegioneEmilia-Romagna

```
    print("\nWaiting for new files...")

observer.join()
```

Viene letta ogni riga contenente i dati degli accelerometri, che viene convertita in un dizionario Python per poi essere trattato come JSON. Il dizionario è così composto:

- id: ID della misurazione corrente (utile a raggruppare delle misurazioni)
- timestamp: timestamp che indica quando la misurazione corrispondente alla riga è stata eseguita; è costruito a partire dal timestamp di inizio della misurazione, sommando per ogni riga il delta time X0
- Xi: valore di X per l'accelerometro i-esimo, con i = 1, 2 o 3,
- Yi: valore di y per l'accelerometro i-esimo, con i = 1, 2 o 3,
- Zi: valore di z per l'accelerometro i-esimo, con i = 1, 2 o 3.

```
{
    "id": "ffa5a7c1-b6ce-4588-a5e5-eb7cbb13cb06",
    "timestamp": "2021-03-19T18:06:21.930733",
    "x1": -0.364879,
    "y1": 0.537611,
    "z1": -0.612346,
    "x2": 0.772346,
    …
}
```

3.1.1 Gateway versione 2

Una nuova versione del Gateway è stata parzialmente testata; si tratta di un modulo Python molto più semplice del precedente, che si occupa semplicemente di inviare un batch di dati a Kafka. Questo è reso possibile grazie alla sua integrazione con LabView, che dalla versione 2018 in poi fornisce nativamente un blocco che consente l'esecuzione di uno script Python, i cui input possono provenire da un qualsiasi altro blocco di LabView. Questa nuova versione del Gateway garantisce prestazioni notevolmente maggiori, per il fatto che i dati non vengono prima salvati su file e poi letti, bensì vengono scambiati direttamente tra due blocchi di codice.

```
from datetime import datetime, timedelta
from uuid import uuid4
import json

from kafka import KafkaProducer

# kafka producer setup
producer = KafkaProducer(bootstrap_servers=["kafkaot:9092"], value_serializer=lambda x:
json.dumps(x).encode('utf-8'))

id = str(uuid4())

def send_data(timestamp, x1, y1, z1, x2, y2, z2, x3, y3, z3, deltatime):
    for i in range(len(x1)):
        # generation of timestamp for keys
        dt = datetime.strptime(timestamp, "%Y/%d/%mT%H:%M:%S,%f") -
timedelta(microseconds=deltatime * pow(10, 6) * (len(x1) - i))

        # for each row, group each sensor values (x, y, z) and send to a Kafka topic (accelerometer)
        accelerometer = {'id': id, 'timestamp': dt.isoformat(timespec="microseconds"), 'x1': x1[i], 'y1':
y1[i], 'z1': z1[i], 'x2': x2[i], 'y2': y2[i], 'z2': z2[i], 'x3': x3[i], 'y3': y3[i], 'z3': z3[i]}
        producer.send('accelerometer_raw', value=accelerometer, key=f'{dt}'.encode())
```

3.2. Kafka e Kafka-Connect

I dati che sono inviati dal Gateway verso Kafka sono considerati dati grezzi, e sono stati quindi organizzati all'interno di un topic denominato *accelerometer_raw*. Attraverso questo topic ogni ipotetico consumatore può leggere ed elaborare i dati grezzi di ognuno dei 3 accelerometri.

Il primo consumatore presente nella pipeline è Kafka Connect, uno strumento appositamente creato per il trasferimento dei dati da e verso Kafka. In modalità *source* funziona da strumento di ingestion di dati verso Kafka, mentre in modalità *sink* funziona come esportatore dati da Kafka verso strumenti esterni. Esiste chiaramente una moltitudine di connettori sink e source disponibili online. Nella pipeline realizzata è stato quindi configurato un connettore Kafka Connect in modalità sink per l'esportazione dei dati verso Elasticsearch.

La configurazione del connettore è mostrata di seguito (in grassetto le impostazioni più rilevanti):

```
{
  "name": "elasticsearch-connector",
  "config": {
    "connector.class": "io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
    "connection.url": "http://elasticsearch:9200",
    "tasks.max": "1",
    "topics": "accelerometer_raw, accelerometer_out",
    "type.name": "_doc",
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "key.converter.schemas.enable" : "false",
    "schema.ignore": "true",
    "flush.timeout.ms": "360000"
  }
}
```

Come si può notare nella configurazione è presente anche un secondo topic, *accelerometer_out*, creato appositamente per contenere gli output delle analisi implementate su MATLAB. In questa maniera è sufficiente configurare un solo connettore per trasferire sia i dati grezzi che gli output delle analisi direttamente su Elasticsearch.

3.4. Matlab

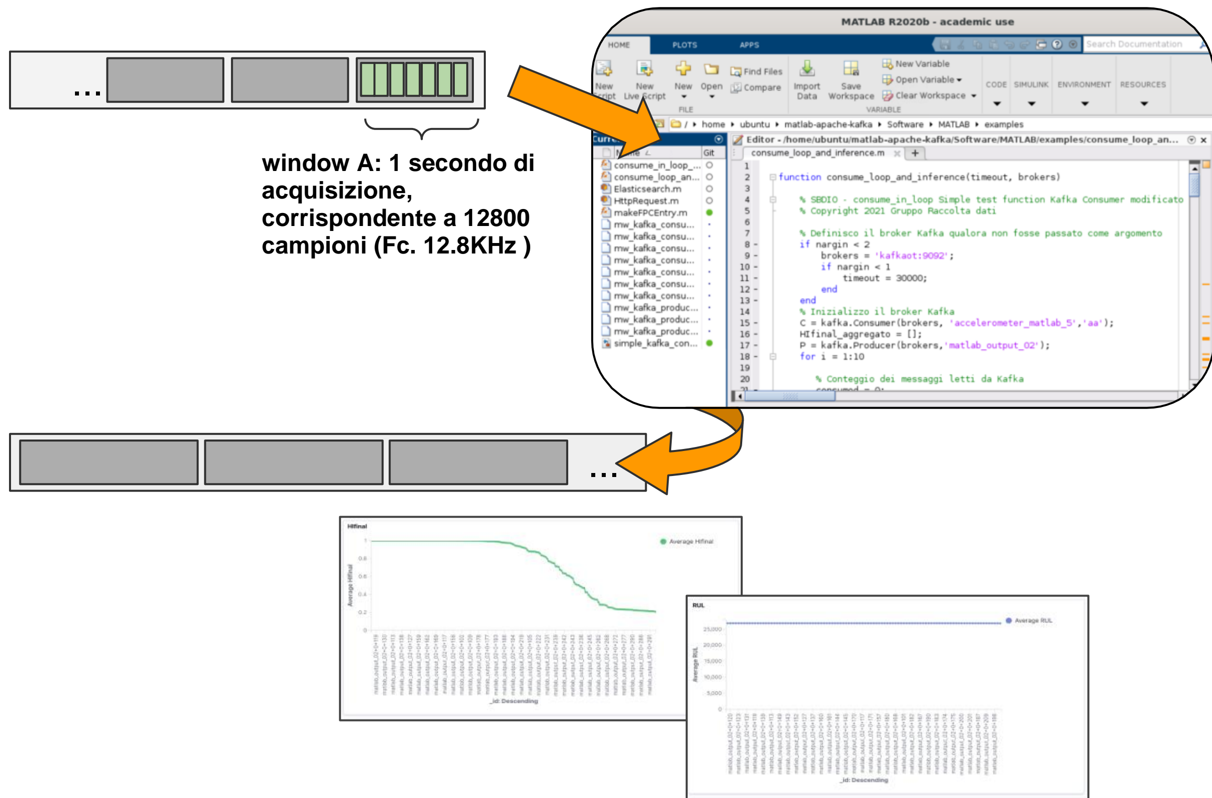


Fig. 5 Integrazione MATLAB - broker Kafka

3.4.1 Codice

```
function consume_loop_and_inference(timeout, brokers)

    % SBDIO - consume_in_loop Simple test function Kafka Consumer modificato
    % Copyright 2021 Gruppo Raccolta dati

    % Definisco il broker Kafka qualora non fosse passato come argomento

    if nargin < 2
        brokers = 'kafkaot:9092';
        if nargin < 1
            timeout = 30000;
        end
    end
```

```
% Inizializzo il broker Kafka

topic_conf = {'auto.offset.reset' 'earliest'};
conf = {};
C = kafka.Consumer(brokers, 'accelerometer_raw','xxx1234',conf,topic_conf);
HIfinal_aggregato = [];
P = kafka.Producer(brokers,'accelerometer_out');

packnominale = zeros(12800,3);

for z = 1:101

  % Conteggio dei messaggi letti da Kafka
  consumed = 0;

  % Quantità max di messaggi da leggere
  batch_size=12800;

  % tempo rimanente prima di arrestare la lettura
  % remaining_timeout = timeout;
  dt = datetime('now');
  dt_end = dt + milliseconds(timeout);

  % Preallocazione della matrice che ospiterà i dati
  output = zeros(12800,3);
  timestamp = 0;
  id = '';
  while consumed < batch_size

     %Leggo un messaggio da Kafka
     [k,v,~] = C.consume();

     % Se il messaggio non è vuoto,
     if ~isempty(v)
        Vl = min(length(v), 400);

        % stampo il messaggio
        %fprintf('%s -- %s\n',k, v(1:Vl));
        consumed=consumed+1;

        % converto il messaggio in array di char
        % e traspongo il vettore, da vettore colonna a riga
        row = char(v(1:Vl)).';

        % decodifico il vettore di char in oggetto JSON
        row = jsondecode(row);

        % estraggo i campi dall'oggetto JSON
        new_row = [row.x1 row.y1 row.z1];

        % inserisco i valori nella riga corrispondente della
        % matrice
        output(consumed,:) = new_row;
```

SBDIO I4.0
Big Data
for Industry

```
            if consumed == 1
               timestamp = row.timestamp;

               % È IMPORTANTE CHE timestamp abbia almeno 1 decimale
               % nei millisecondi: ad esempio timestamp = '2021-05-25T09:00:19.6'
               date = datetime(timestamp,'InputFormat',  \
                   'yyyy-MM-dd"T"HH:mm:ss.SSSSS', \
                  'Format','yyyy-MM-dd"T"HH:mm:ss.SSSSS');
            end
            id = row.id;
         end
         % se il tempo rimasto è scaduto, termino il ciclo
         % perchè evidentemente non ci sono più messaggi da leggere
         remaining_timeout = milliseconds(dt_end - datetime('now'));
         if remaining_timeout <= 0
            fprintf('Timeout scaduto');
            return;

         end
     end

    fprintf('Lettura finestra %d ok',z);

    if z==1
       %packnominale = output(1:6400,:); %input nominale, resta sempre lo stesso
       packnominale = output;
       continue
    end
    % packdeteriorato = output(6400:end,:) ;
    packdeteriorato = output;
    %packnominale=[packnominale(1:16000000,:)]; potrebbe essere necessario
    %adottare uno stesso numero di elementi per non andar incontro a problemi
    %di incompatibilità successivi

    %packdeteriorato=[data_streaming_kafka_successivi];%input deteriorato
    meann=mean(packnominale); %valor medio nominale
    meand=mean(packdeteriorato);  %valor medio deteriorato

    % Marco
    % AMMETTIAMO un dataset totale di 12800 campioni, che suddividiamo in 10 blocchi da 1280
    % packnominale è il 1° blocco da 1280
    % packdeteriorato sarebbero i restanti 9 blocchi, da 9 x 1280 campioni

    %INIZIALIZZO

    HInumeratore=zeros(length(packdeteriorato),3); %inizializzazione dimensioni matrici
    HIdenominatore1=zeros(length(packdeteriorato),3);
    HIdenominatore2=zeros(length(packdeteriorato),3);

    % 6400 = 12800 / 2
    % Marco
    % come dimensione blocco metto 640 = 1280 / 2
    His=zeros(round((length(packdeteriorato))/1280),3);
```

```
    Hii1=zeros(round((length(packdeteriorato))/1280),3);
    Hii2=zeros(round((length(packdeteriorato))/1280),3);
    HI=zeros(round((length(packdeteriorato))/1280),3);
    idx=1:1280:length(packdeteriorato);
%definisco un vettore che va di 6400 in 6400 sino al valore totale delle samples acquisite , questo
vettore mi serve per eseguire le sommatorie ogni 6400 campioni ed ottenere quindi un valore di HI
per ogni 0.5 secondi, ovviamente questo processo è modificabile variando il numero dei samples
da considerare per ogni sommatoria

    %HI

    for j=1:3
        % disp(j);
        for i=1:length(packdeteriorato)-1
        % disp(i);
        HInumeratore(i,j)=((packnominale(i,j)-meann(j)).*(packdeteriorato(i,j)-meand(j)));
HIdenominatore1(i,j)=((packnominale(i,j)-meann(j))^(2));
        HIdenominatore2(i,j)=((packdeteriorato(i,j)-meand(j))^(2));
        end
    end

    for j=1:3
    for k=1:length(idx)-1
        His(k,j)=sum(HInumeratore(idx(k):idx(k+1)-1,j));
        Hii1(k,j)=sum(HIdenominatore1(idx(k):idx(k+1)-1,j));
        Hii2(k,j)=sum(HIdenominatore2(idx(k):idx(k+1)-1,j));
        HI(k,j)=abs((His(k,j))/(sqrt(((Hii1(k,j))*(Hii2(k,j))))));
    end
    end


    %Smoothing (simple moving average)

    %smoothed=transpose(movmean(HI,[9 0]));
    HIs=zeros(length(HI));
    HIs=movmean(HI,[3 0]);
    HIfinal=zeros(length(HIs),1);
    %his=HIs(:,1:9);

    %Eseguo media degli HI su assi per ricondurmi ad un unico HI
    r = length(HIs);
    for s=1:length(HIs)
    %disp(s)
    HIfinal(s)=(1/3)*sum(HIs(s,:));
    end
    %La matrice HIfinal è 10x1: ogni riga corrisponde
    %ad 1 decimo di secondo di dati (ovvero 12800 / 10 campioni)

    HIfinal;
    HIfinal_aggregato = [HIfinal_aggregato;HIfinal];
    HIfinal_aggregato;

    % HIfinal_aggregato=zeros(20,1);%somme degli HIfinal fino a questo momento
```

```
    x=transpose(1:1:length(27434));
    fittingcurve=0.04178*exp((-2.647e-05)*x);

    for s=1:length(HIfinal)

        tfinal=49.4747;
        rul(s)=tfinal-(HIfinal(s))^(-1); %[s]
        RUL(s)=(rul(s))*27434 /(tfinal);
        date_1 = date + seconds(0.1)*(s-1);

        % {"HIfinal":"0.74789","RUL":"26692.5776","timestamp":"2021-05-20T11:41:43.555"}
        date_1_str = datestr(date_1,"yyyy-mm-ddTHH:MM:ss.FFF")
        key = strcat(id,'-',num2str((z-1)*10+s));
        S = struct('HIfinal',HIfinal(s),'RUL',RUL(s),'timestamp',date_1_str)
        P.publish(key,jsonencode(S));
    end

    end
end
```

3.5. Nuove visualizzazioni su Kibana

Si tratta del componente che offre agli utenti, anche quelli non tecnici, di creare visualizzazioni sui dati salvati ed indicizzati in Elasticsearch.

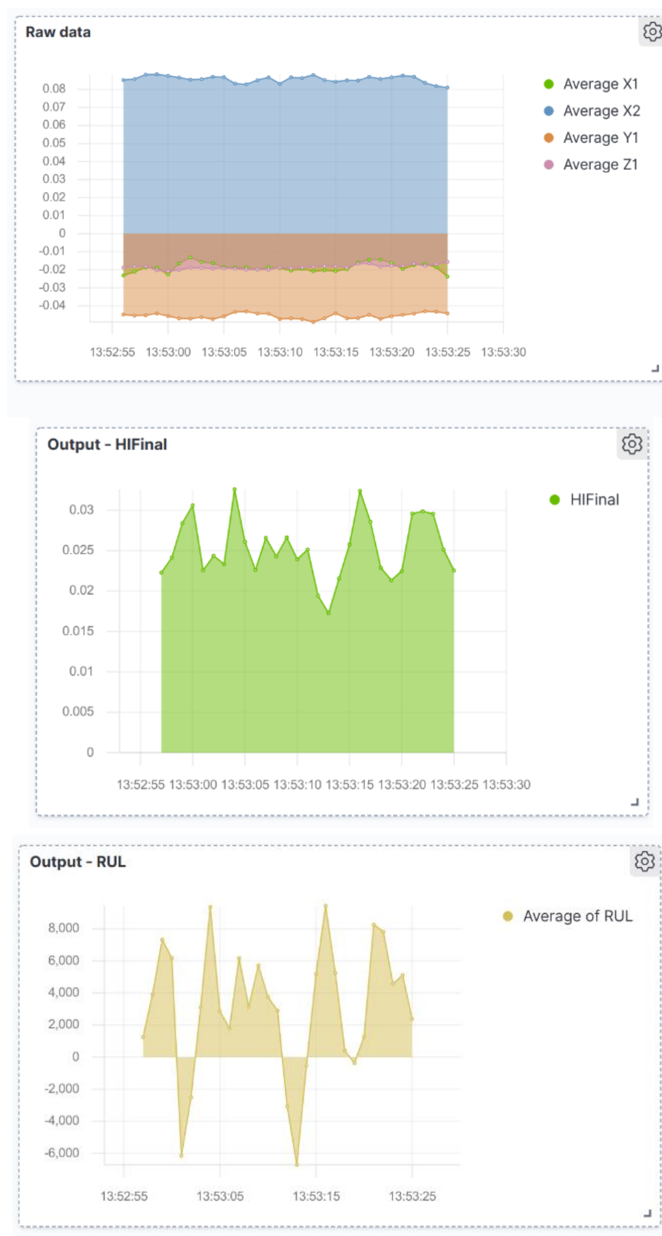


Fig. 6 Visualizzazioni su Kibana

4. Deploy, esecuzione e migrazione dei servizi

Inizialmente per una questione di test, il deploy dello stack è stato eseguito nel cloud di UniBo; successivamente lo stack è stato migrato nel cloud di T3Lab.

4.1. Deploy dei servizi su CIRI-ICT

Il deploy dei servizi è stato effettuato su un cluster Kubernetes (gestito con Rancher) messo a disposizione dall'Università di Bologna. I servizi in questione sono quelli già citati:

- Layer di Ingestion e Messaging gestito con Kafka e Kafka Connect
- Layer di Storage gestito con Elasticsearch
- Layer di Visualization gestito con Kibana

La configurazione di Kafka è molto rapida, infatti i topic che contengono i dati si creano in automatico all'arrivo dei dati stessi; quella di Kafka Connect ha previsto l'aggiunta del plugin di Elasticsearch (open source e scaricabile direttamente dal sito di Elasticsearch: https://www.confluent.io/hub/confluentinc/kafka-connect-elasticsearch) e la sua conseguente configurazione tramite un file JSON come descritto nella documentazione.

Elasticsearch non ha necessitato di configurazioni particolari, in quanto è stata riposta attenzione al formato dei dati in ingresso, in modo tale che riconoscesse automaticamente il tipo dei dati (stringhe, interi e timestamps).

Per quanto riguarda Kibana, è stato discusso con i diretti utilizzatori come i grafici dovessero apparire e quali dati dovessero mostrare; da questi confronti sono state create varie dashboard, alcune riportate come screenshot nell'apposito paragrafo precedente.

Per l'esecuzione della pipeline, gli step da seguire sono semplici:

- Avvio del Gateway Python con un doppio click sulla macchina Windows presente in laboratorio (con il nuovo gateway questa operazione non è necessaria, perché direttamente integrato nello script LabView)
- Avvio dello script Matlab per l'elaborazione dei dati in ingresso a Kafka
- Apertura di Kibana tramite browser per osservare i grafici presenti sulle dashboard
- Avvio delle misurazioni di test tramite LabView

Nel giro di qualche secondo, i nuovi dati grezzi e i nuovi dati elaborati iniziano a fluire nei grafici.

4.2. Migrazione del cluster su T3Lab

La migrazione dello stack nel cloud di T3Lab è stata velocissima. Per quanto riguarda i servizi, è bastato semplicemente far avere ai tecnici di T3Lab la lista delle tecnologie necessarie e i loro file di configurazione, che sono completamente identici a quelli usati per CIRI-ICT.

Una volta effettuato il deploy e la configurazione dei servizi da parte di T3Lab, gli ultimi step richiesti per mettere in moto la pipeline sono stati:

- modificare il destinatario dei dati uscenti dal gateway, dal broker Kafka presente nel cloud di CIRI-ICT a quello presente nel cloud di T3Lab.
- dato che non è stato possibile spostare Matlab per una questione di licenze (è dovuto rimanere all'interno di UniBo), è stato modificato il destinatario dei dati elaborati, dal broker Kafka di CIRI-ICT a quello di T3Lab.

Anche qui, una volta completati correttamente questi step che hanno richiesto davvero poco tempo, nel giro di qualche secondo i dati grezzi ed elaborati hanno iniziato a fluire nella pipeline.