Architettura del progetto

Si procede con l'illustrazione degli obiettivi del progetto e le connesse modalità di realizzazione. Dapprima saranno presentati intenti e architettura in maniera generica, in seguito si scenderà nel dettaglio per poter approfondire ogni singolo aspetto del progetto. In particolare, si è pensato di suddividere la realizzazione effettiva del progetto in tre fasi, ognuna delle quali rappresenta una parte fondamentale del processo di elaborazione dei dati: dalla loro generazione, passando dalla validazione, fino alla raffinazione.

Come indicato in Figura 1, l'obiettivo del progetto consiste nella realizzazione di una data pipeline nella quale i dati vengono processati in più fasi da diverse tasks, al fine di poter verificarne la correttezza e rilevare, conseguentemente, l'eventuale presenza di problemi all'interno dei sistemi in questione.

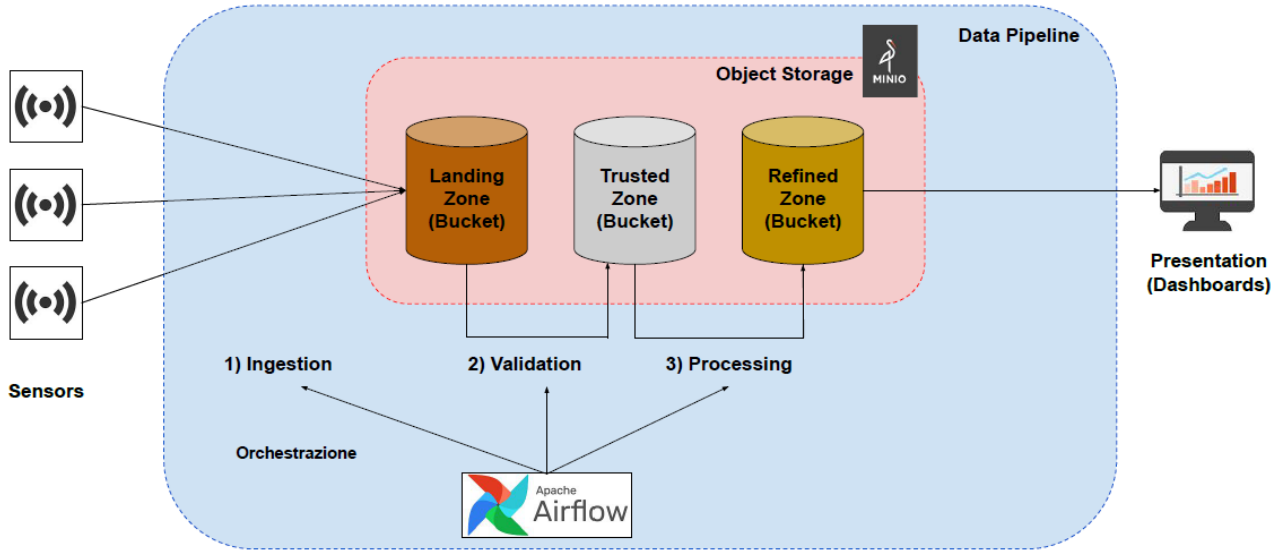

Figura 1: Architettura del progetto

L'architettura creata è composta da uno script in linguaggio di programmazione Python che simula la generazione dei dati provenienti da un determinato numero di sensori e poi li trasmette a uno specifico contenitore preparato per ospitarli, chiamato bucket, situato all'interno di MinIO, un Object Storage, al fine di assicurarne una duratura archiviazione (fase uno: Ingestion). A questo punto, però, i dati grezzi, così come creati dai dispositivi di provenienza, possono presentare errori di rilevamento o un presentarsi con dato corrotto, motivo per cui bisognerà trattare l'eventuale malformazione dei dati attraverso una validazione. Infatti, quando i dati vengono recuperati dal precedente bucket, vengono esaminati uno a uno e, solo una volta verificate la loro integrità e validità, vengono ritrasmessi a un secondo bucket dello stesso MinIO. Quest'ultimo è un contenitore analogo al

precedente, ma con la peculiarità di essere l'unica fonte di verità dei dati (fase due: Validation). L'ultima fase prevede il recupero dei dati convalidati e ritenuti affidabili ottenuti tramite la precedente fase, un ulteriore controllo che permette di eseguire operazioni specifiche al caso d'uso in considerazione, quindi i dati aggiornati vengono salvati in formato strutturato e inviati a un terzo e ultimo bucket, sempre su MinIO. Quest'ultimo è un contenitore analogo ai precedenti, ma con il vantaggio di possedere dati completamente processati, strutturati e pronti all'immediato uso futuro in caso di necessità da parte di applicazioni esterne (fase tre: Processing).

Queste tre fasi sono scandite da una pianificazione di data pipelines gestita totalmente da Apache Airflow, con intervalli differenti per l'esecuzione delle varie tasks in relazione al loro ruolo, le cui attese ed esecuzioni risultano costantemente monitorate e all'occorrenza garantite dallo scheduler. Tramite il webserver di Airflow, avviabile in locale, invece, è possibile attivare le DAG d'interesse perché siano a tutti gli effetti considerate in esecuzione e quindi di rilevanza per lo scheduler, il quale se ne occuperà al momento opportuno. Infine, anche per il servizio di Object Storage offerto da MinIO è necessario attivare un webserver, avviabile in locale.

Preparazione e configurazione di Apache Airflow

Per poter utilizzare Apache Airflow bisogna prima specificare la necessità di preparare un ambiente adatto ad ospitare i vari componenti di Airflow. Per impostare l'ambiente (environment), Airflow si appoggia a varie soluzioni, tra cui Docker, il Python Virtual Environment o Kubernetes Namespaces: nel progetto viene adottato l'ambiente virtuale Python.

Il primo step consiste nel procedere con l'installazione di Anaconda e la creazione di un ambiente virtuale personalizzato. È, infatti, possibile creare un ambiente virtuale che andrà a contenere tutti i componenti di Airflow grazie all'ausilio di Conda, che permette di generare e caricare in velocità e con facilità vari virtual environments. Conda, sistema di gestione di pacchetti open-source e gestore di ambienti virtuali, è incluso in tutte le versioni di Anaconda e Miniconda. Anaconda è scaricabile dal sito ufficiale https://www.anaconda.com/download, mentre il virtual environment si può creare o eliminare e una volta creato attivare o disattivare. Naturalmente, è anche possibile visualizzare l'elenco degli ambienti disponibili.

Attenzione: è estremamente importante ricordare che non tutte le versioni dell'ambiente Python sono interamente compatibili con le varie versioni dei tools che verranno adottati, come Airflow, Pandas o fastparquet. Si raccomanda fortemente di adottare l'installazione di un ambiente virtuale Python versione 3.7, come suggerito dai seguenti comandi d'esempio.

Seguono i comandi per creare ed eliminare un ambiente virtuale Python versione 3.7 dal nome “python_virtual_env” e il comando per visualizzare l’elenco di tutti gli ambienti disponibili:

```
conda create -name python_virtual_env python=3.7 -y

conda remove -n python_virtual_env --all

conda env list
```

Seguono i comandi per attivare o disattivare l’ambiente virtuale Python dal nome “python_virtual_env”:

```
conda activate python_virtual_env

conda deactivate
```

Una volta attivato l’ambiente virtuale, si può procedere con l’installazione di Apache Airflow, guidata dal Quick Start nel sito ufficiale di Airflow [24], offerta dal repository Python Package Index (PyPI). Si noti che il comando sfrutta il constraint file determinato dallo specifico URL così pensato:

```
https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/constraints-${PYTHON_VERSION}.txt
```

Dopo aver verificato la compatibilità tra la versione di Airflow e la versione dell’ambiente virtuale Python, si può quindi procedere davvero con l’installazione di Apache Airflow. Grazie al comando che segue, viene installata la versione 2.6.0:

```
pip install "apache-airflow==2.6.0" -constraint https://raw.githubusercontent.com/apache/airflow/constraints-2.6.0/constraints-3.7.txt
```

È possibile verificare se l’installazione è riuscita in maniera del tutto corretta interrogando Airflow sull’attuale versione con il comando:

```
airflow version
```

A questo punto si può procedere con l’inizializzazione del database di Airflow col comando:

```
airflow db init
```

Quindi, prima di lanciare il web server, è consigliato verificare l’esistenza di un account per effettuare proprio il futuro accesso al web server. Airflow crea un account di default, di cui si può verificare l’esistenza con il comando:

```
airflow users list
```

In alternativa, si può procedere con la creazione di un account tramite un apposito comando che permette di definire numerosi campi:

```
airflow users create \
--username AdminDiProva \
--password AdminDiProva \
--firstname Nome \
--lastname Cognome \
--role Admin \
--email Nome.Cognome@example.org
```

Ora è possibile lanciare il webserver di Airflow, per poi poter finalmente procedere all'autenticazione presso l'indirizzo locale definito dalla particolare porta su cui lanciamo il server. Airflow consiglia il seguente comando, in cui si può comunque valutare l'omissione della specifica della porta da adottare:

```
airflow webserver --port 8080
```

Infine, ultimo ma non di importanza, per mantenere i DAG sempre aggiornati rilevando le ultime modifiche o le nuove aggiunte, è possibile avviare lo scheduler tramite il comando:

```
airflow scheduler
```

Poiché si è parlato dell'installazione e configurazione di Apache Airflow, si riportano di seguito altri comandi di frequente utilizzo: il primo permette di visualizzare la lista di tutti DAG che riportano un qualsiasi tipo di errore nel codice, il secondo permette di visualizzare la lista di tutti i DAG che non presentano errori nel codice, il terzo permette di eseguire manualmente il DAG specificato, nell'esempio chiamato "my_first_dag", per l'operazione di testing.

```
airflow dags list-import-errors

airflow dags list

airflow dags test my_first_dag
```

Attenzione: è di estrema importanza notare che per rendere visibili i DAG scritti manualmente affiancandoli ai già preesistenti DAG d'esempio offerti da Airflow, è

indispensabile esportare la directory di Airflow impostando, tramite comando, il percorso della cartella di Airflow (di default il nome della directory è “airflow”) presente nella nostra macchina. Il comando da adottare nel caso del presente elaborato è quello che segue:

```
export AIRFLOW_HOME="/home/user/airflow"
```

Si vuole far notare il file di configurazione di Airflow, “air flow.cfg” di default presente nell’omonima cartella, per procedere con un’esperienza di personalizzazione completa.

Si noti che la configurazione di Airflow è completa, ma è necessario fare menzione anche dei comandi necessari per settare il corretto funzionamento di MinIO, l’Object Storage. A tal proposito,è sufficiente procedere con i comandi proposti nella sezione “download” del sito ufficiale di MinIO, all’indirizzo https://min.io/. I comandi adottati nel presente progetto sono i seguenti:

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio

chmod +x minio

MINIO_ROOT_USER=admin MINIO_ROOT_PASSWORD=password ./minio
server /mnt/data --console-address ":9001"
```

L’ultimo comando rimane di valido utilizzo ogni qual volta futura si desideri rilanciare il webserver di MinIO.

Si consiglia anche di utilizzare il seguente comando specifico di Python per l’installazione del Software Development Kit (SDK):

```
pip install minio
```

Infine, il progetto descritto nel presente elaborato necessita dell’installazione di altri due importanti strumenti: Pandas (Python Data Analysis Library) e fastparquet. Per procedere all'installazione di Pandas, è possibile utilizzare il comando:

```
conda install pandas
```

Per installare fastparquet [25], invece, utilizzare il comando che segue:

```
pip install fastparquet
```

Documentazione fase 1: Data Ingestion

La prima fase dell'orchestrazione, Data Ingestion, viene trattata da Apache Airflow tramite un DAG, denominato "data_ingestion.py", scritto in linguaggio di programmazione Python. Questo step prevede la generazione simulata di dati provenienti da dispositivi connessi in rete che, per poter svolgere in maniera efficiente, sicura e affidabile i propri compiti, necessitano l'obbligo di un continuo monitoraggio del corretto funzionamento attraverso la trasmissione costante di dati che tengano traccia dello stato dei dispositivi stessi. I suddetti dati vengono processati nel DAG sopracitato attraverso l'esecuzione di due differenti tasks, pianificate, nel presente elaborato, per essere eseguite ogni ora del giorno a partire dalla mezzanotte del 2023/06/13.

Si richiama l'attenzione sui concetti di data interval (intervallo temporale) e logical date sui quali si basa lo scheduling di Airflow. Infatti, se la pianificazione prevede l'esecuzione del DAG ogni ora, significa che verrà eseguito "allo scadere di ogni intervallo di un'ora a partire dalla mezzanotte", con conseguente prima esecuzione alle ore 01:00 (a.m.); oppure, se è prevista l'esecuzione del DAG ogni sei ore, significa che verrà eseguito "allo scadere di ogni intervallo di sei ore a partire dalla mezzanotte", con conseguente prima esecuzione alle ore 06:00 (a.m.). Questo appena esplicato data interval differisce dalla logical date, che rappresenta, invece, l'orario di inizio del conteggio dell'intervallo temporale pianificato (data interval), ovverosia mezzanotte in entrambi i casi prima citati. Per poter gestire agevolmente gli scheduling più complessi, si fa menzione di un prezioso tool online disponibile per visualizzare a parole l'espressione di pianificazione desiderata, nonché la prossima esecuzione: https://crontab.guru/.

Nella Figura 2 che segue, viene riportato il DAG "data_ingestion.py", il quale necessita di fondamentali import per poter funzionare in maniera corretta.

```
from airflow import DAG

from airflow.operators.python import PythonOperator
from datetime import datetime


with DAG ('data_ingestion', start_date=datetime(2023,6,13), schedule='0 */1 * * *') as dag:

        data_creator=PythonOperator(
        task_id='data_creator',
        python_callable=dct)

        data_loader=PythonOperator(
        task_id='data_loader',
        python_callable=dlt)


data_creator >> data_loader
```

Figura 2: Il DAG "data_ingestion" e le relative due tasks

Prima di procedere con la descrizione delle due tasks, si vuole richiamare l'attenzione sull'ultima riga di codice, di fondamentale importanza. Tale scrittura sfrutta l'operatore ">>", che in Airflow ha lo scopo di specificare le dipendenze tra le tasks all'interno del DAG, indicando l'ordine di esecuzione: solo una volta completata l'attività "data_creator", si procederà con l'esecuzione di "data_loader".

La prima task, definita nel file "data_creator.py", realizza la simulazione dei dati come sopra descritta attraverso l'utilizzo di un file in formato testuale denominato "sensors_data.txt". Lo script crea una struttura dati composta da quattro campi: ID del sensore, data e ora al momento della simulazione del rilevamento, temperatura trasmessa e unità di misura relativa alla temperatura. In questo ordine, i dati vengono composti in una riga singola seguita da un break line per ciascun sensore. Il valore della temperatura potrebbe risultare danneggiato al momento della generazione, caso facilmente riconoscibile dalla presenza della dicitura "null" al posto del valore numerico presso il campo della temperatura.

La seconda task, definita nella funzione "file_uploader" all'interno del file "minio_file_manager.py", realizza il caricamento dei dati originati nella precedente attività caricando il file "sensors_data.txt" all'interno di un bucket all'interno di MinIO chiamato "landingzone". Lo script è pensato per stabilire una connessione presso il server MinIO, quindi verificare l'esistenza del bucket, provvedendo alla sua creazione nel caso in cui non esistesse. È fondamentale ricordare che questo container accoglie dati che possono presentare errori al momento della generazione, dati che non possono essere utilizzabili senza validazione perché inaffidabili. Si noti che a ogni esecuzione pianificata il file viene caricato con tutti i dati rilevati dall'attivazione del DAG, riportando quindi tutte le generazioni di dati dovute alle esecuzioni pregresse. Nella seguente Figura 3 è riportato il codice che definisce la seconda task, ovverosia la funzione "file_uploader" all'interno del file "minio_file_manager.py", richiamando l'attenzione alla funzione "client.fput_object()".

```
from minio import Minio


def funct_uploader(namebucket, namefile, datafolder):

        # Create a client with the MinIO server playground,
        #its access key and secret key
        client = Minio(
                "localhost:9000",
                access_key="admin",
                secret_key="password",
                secure=False,)

        # Make 'landingzone' bucket if not exist
        found = client.bucket_exists(namebucket)
        if not found:
                client.make_bucket(namebucket)

        # Upload '/home/giacomo/Documenti/Letture_sensori/sensors_data.txt'
        # as object name 'sensorsdata.txt' to bucket namebucket
        client.fput_object(namebucket, namefile, datafolder + namefile,)
        print(datafolder + namefile + " is successfully uploaded as "
                "object '" + namefile + "' into bucket '" + namebucket +"'.")
```

Figura 3: Codice della funzione "funct_uploader" definita nel file "minio_file_manager.py"

Documentazione fase 2: Data Validation

La seconda fase dell'orchestrazione, Data Validation, viene trattata da Apache Airflow tramite un DAG, denominato "data_validation.py", scritto in linguaggio di programmazione Python. Questo step prevede il recupero dei dati generati dalla precedente fase, la loro validazione e, infine, la loro archiviazione. I suddetti dati vengono processati nel DAG sopracitato attraverso l'esecuzione di tre differenti tasks, pianificate, nel presente elaborato, per essere eseguite ogni tre ore del giorno a partire dalla mezzanotte del 2023/06/13. Nella Figura 4 che segue, viene riportato il DAG "data_validation.py" (da ora si omettono gli import del caso).

```
with DAG('data_validation', start_date=datetime(2023,6,13), schedule='0 */3 * * *') as dag:

        data_downloader=PythonOperator(
        task_id='data_downloader',
        python_callable=ddt)

        data_validator=PythonOperator(
        task_id='data_validator',
        python_callable=dvt)

        data_uploader=PythonOperator(
        task_id='data_uploader',
        python_callable=dut)


data_downloader >> data_validator >> data_uploader
```

Figura 4: Il DAG "data_validation" e le relative tre tasks

La prima task, definita nella funzione "file_downloader.py" all'interno del file "minio_file_manager.py", realizza il recupero dei dati originati nella precedente fase attraverso il download del file "sensors_data.txt" dal bucket "landingzone" di MinIO. Non verrà riportato il codice di tale attività, in quanto le due funzioni adottano la stessa tipologia di funzione: infatti, le due differiscono solamente per l'azione compiuta, "put" o "get". La prima, put, sulla quale abbiamo richiamato l'attenzione poco prima della figura 3, realizza l'upload o caricamento del dato; la seconda, get, realizza il download o scaricamento del dato.

La seconda task, definita nel file "file_validator.py", realizza la validazione dei dati presenti all'interno del file "sensors_data.txt". Lo script verifica, sensore per sensore, riga per riga, se il dato presenta valori corrotti presso il campo della temperatura, quindi procede a validarlo sostituendo il dato danneggiato con la stringa d'informazione "ERRORE DI RILEVAMENTO". Si noti un'ulteriore questione: per evitare di processare tutti i dati a ogni esecuzione del DAG e per evitare informazioni duplicate successive alla validazione, si è

pensato di esaminare solamente i dati generati nelle precedenti tre ore dal momento di esecuzione del DAG. In questo modo, vengono controllati esattamente i dati generati nell'ultimo intervallo di tempo, i quali, oggetto della pianificazione ogni tre ore, rappresentano gli ultimi e unici valori che non sono ancora stati esaminati. I dati così processati vengono quindi salvati in formato json all'interno del file "sensors_data.json", pronti per essere archiviati nella terza task. Il metodo adottato per la memorizzazione dei dati in tale formato è json.dumps(), offerto dal pacchetto json facilmente importabile in Python.

Il codice della task prevede dapprima il recupero della precedente lista di dati, qualora già esistente, quindi procede con l'estensione delle informazioni aggiungendovi solamente i dati generati nelle tre ore precedenti all'esecuzione. Infine, lo script memorizza il risultato su un file in formato json sfruttando le funzioni del caso, come anticipato poc'anzi. Nella Figura 5, è riportato un estratto che mostra la lettura del file "sensors_data.txt", il controllo temporale sulla trascorsa generazione dei dati e il finale salvataggio su file.

```
if (check == True):
        # Se il file esiste si procede con la lettura del file txt per salvare i dati in una lista
        # che sarà appesa ad Array (eventualmente già riempito della preesistente lista)
        with open(argpath + oldnamefile, 'r', encoding='UTF-8') as file:
                for line in file:
                        array = line.rstrip().split(",")
                        now = datetime.now()
                        generationtime = array[1]
                        delta = now - datetime.strptime(generationtime, '%Y-%m-%d %H:%M:%S.%f')
                        hours = (delta.total_seconds() / 60) / 60
                        # Se il Timestamp risale a meno di tre ore recupero i dati
                        if (1 <= int(hours) <= 3):
                                if (array[2] == "null"):
                                        array[2] = "TEMPERATURE DETECTION ERROR"
                                        corrupted_lines += 1

                                # Appendo su Array vuoto o sui precedenti dati
                                Array.append(array)

        # Scrivo il nuovo file e ritorno il numero di righe modificate
        jsonString = json.dumps(Array)
        jsonFile = open(argpath + newnamefile, "w", encoding='UTF-8')
        jsonFile.write(jsonString)
        jsonFile.close()
```

Figura 5: Estratto della seconda task definita in "file_validator.py"

La terza task, definita nella funzione "file_uploader" all'interno del file "minio_file_manager.py", realizza il caricamento dei dati elaborati nella precedente attività trasmettendo il file "sensors_data.json" all'interno di un bucket all'interno di MinIO chiamato "trustedzone". Lo script è pensato per stabilire una connessione presso il server MinIO, quindi verificare l'esistenza del bucket, provvedendo alla sua creazione nel caso in cui non esistesse. Si noti che la funzione in questione è esattamente la stessa utilizzata dalla seconda task all'interno della fase uno, in quanto il formato dei dati nel file da archiviare e il nome del bucket che ospiterà i dati sono parametri trasmessi dal file che definisce il DAG e

non propri della funzione definita per l'esecuzione della task. Per questo stesso motivo, non verrà riportato il relativo codice, che rimane visibile nella Figura 3. È fondamentale ricordare che questo bucket accoglie dati che possono essere considerati attendibili e validi a tutti gli effetti, dati che possono essere utilizzati in quanto resi affidabili grazie alla fase di validazione.

Documentazione fase 3: Data Processing

La terza fase dell'orchestrazione, Data Processing, viene trattata da Apache Airflow tramite un DAG, denominato "data_processing.py", scritto in linguaggio di programmazione Python. Questo step prevede il recupero dei dati generati dalla precedente fase, un loro ulteriore processamento e, infine, la loro archiviazione. I suddetti dati vengono processati nel DAG sopracitato attraverso l'esecuzione di tre differenti tasks, pianificate, nel presente elaborato, per essere eseguite ogni sei ore del giorno a partire dalla mezzanotte del 2023/06/13. Nella Figura 6 che segue, viene riportato il DAG "data_processing.py".

```
with DAG('data_processing', start_date=datetime(2023,6,13), schedule='0 */6 * * *') as dag:

        data_downloader=PythonOperator(
        task_id='data_downloader',
        python_callable=ddt)

        data_processor=PythonOperator(
        task_id='data_processor',
        python_callable=dpt)

        data_uploader=PythonOperator(
        task_id='data_uploader',
        python_callable=dut)


data_downloader >> data_processor >> data_uploader
```

Figura 6: Il DAG "data_processing" e le relative tre tasks

La prima task, definita nella funzione "file_downloader.py" all'interno del file "minio_file_manager.py", realizza il recupero dei dati originati nella precedente fase attraverso il download del file "sensors_data.json" dal bucket "trustedzone" di MinIO. Ancora una volta, si noti che questa funzione è la stessa impiegata nella prima task della fase due.

La seconda task, definita nel file "file_processor.py", realizza un ulteriore processamento dei dati presenti all'interno del file "sensors_data.json". Lo script carica i dati grazie al metodo json.load(), offerto dal pacchetto json importabile in Python. Lo script verifica i dati, sensore per sensore, corredando di ulteriore campo la struttura di ogni gruppo di dati relativi a un

determinato sensore, dopo aver verificato o meno una determinata condizione. In particolare, il campo aggiunto prende il nome di “Warning”, e può avere due soli valori: “OK” oppure “ALERT”. I dati così processati vengono quindi salvati in formato parquet, in modo tale che all’interno di un determinato file col nome di un sensore, per esempio “ID000001.parquet”, ci siano tutti e i soli dati misurati da quello specifico sensore in questione. Il metodo adottato per la memorizzazione dei dati in tale formato è pd.DataFrame().to_parquet(): “pd.DataFrame()” è offerto dalla libreria Python Data Analysis Library (pandas) e permette di trasformare i dati in oggetto in dataframe, mentre la funzione “to_parquet()” consente la scrittura dei dataframe in formato binario di tipo parquet. Quindi, generato un singolo file per ogni differente sensore, i dati in formato parquet sono pronti per essere archiviati nella terza task.

Il codice della task prevede dapprima il recupero dei dati salvati in formato json, poi la modifica dei dati con l’introduzione del sopracitato campo “Warning”, quindi procede con il salvataggio dei dati generati da ciascun sensore nel rispettivo omonimo file in formato parquet. Nella Figura 7, è riportato un estratto del file “file_processor.py”che mostra esattamente l’ultimo step di quanto appena descritto, ovverosia il salvataggio delle informazioni nel formato finale.

```
counter = 0
while(counter < len(SortedArray)):
        LastArray = []
        sensorid = SortedArray[counter][0]
        while(counter < len(SortedArray) and sensorid == SortedArray[counter][0]):
                LastArray.append(SortedArray[counter])
                counter += 1
                dataframe = pd.DataFrame(LastArray)
        dataframe.columns = ["SensorID","Timestamp","Temperature","Unit","Warning"]
        print(dataframe)
        dataframe.to_parquet(argpath + sensorid + '.parquet')
```

Figura 7: Estratto della seconda task definita in “file_processor.py”

La terza task, definita nella funzione “file_uploader” all’interno del file “minio_file_manager.py”, realizza il caricamento dei dati elaborati nella precedente attività caricando i file in formato parquet all’interno di un bucket all’interno di MinIO chiamato “refinedzone”. Lo script è pensato per stabilire una connessione presso il server MinIO, quindi verificare l’esistenza del bucket, provvedendo alla sua creazione nel caso in cui non esistesse. Si noti che la funzione in questione è esattamente la stessa utilizzata dalla seconda task all’interno della fase uno e dalla terza task all’interno della fase due, in quanto il formato dei dati nel file da archiviare e il nome del bucket che ospiterà i dati sono parametri trasmessi dal file che definisce il DAG e non propri della funzione definita per l’esecuzione della task. È fondamentale ricordare che questo contenitore accoglie dati che si presentano del tutto pronti per l’uso in caso di necessità da qualsiasi applicazione esterna.