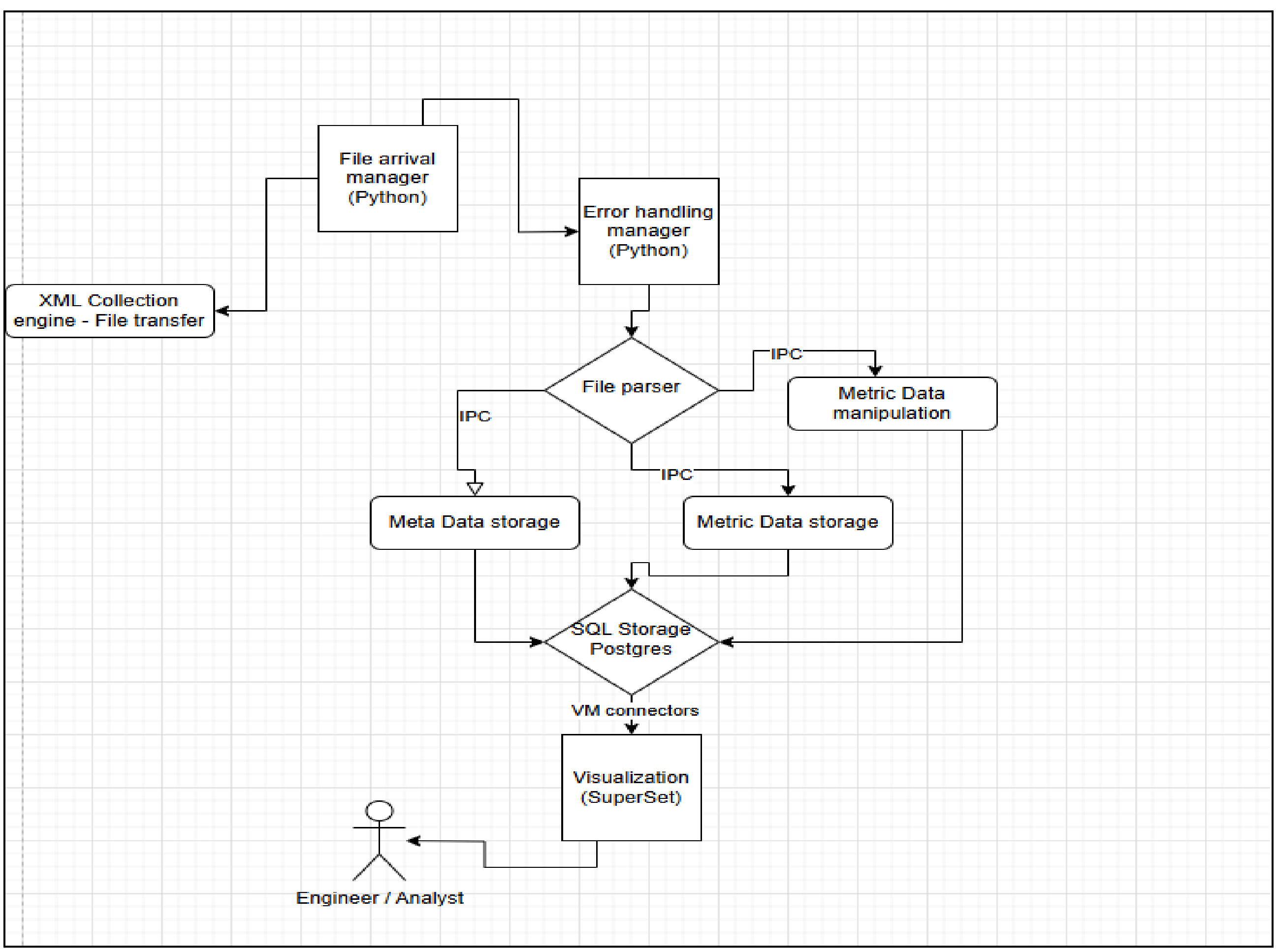
Automated Data Processing and Database Management Pipeline for Data Analytics

Lars Finlayson

University of Pennsylvania

Project Documentation:

In many systems, hierarchical levels of data collection need to be examined to get a full picture of performance. My project was a wireless system divided into controllers and access points (APs). The goal was to structure the data so engineers and technicians could analyze it for long-term trends and short-term anomaly events. The complete environment for this data contained 1000s of controllers, each managing 100s of access points. Each element created 1000s of data points every 15 minutes that needed to be manipulated for storage and to maintain their time series structure. To allow the project to scale to handle this potential size a combination of virtual machines(VMs) and virtual machine containers (docker) was built to allow resources to be added to individual tasks efficiently. My audience is only some Python literate, so selecting a visualization tool that would be understandable to that group is very important for everyone looking at the data.



<u>Problems and Obstacles</u>:

When first looking at the code, converting the XML file straight into the SQL database would not work due to how the data was structured inside the file. The code needed to be converted from XML to CSV and then processed automatically into the database as this was a continuously growing file. The next problem was making the code robust enough to ensure it wouldn't break if an XML file formatted incorrectly, missing data, corrupted, or any other number of problems that can break the code didn't make it through the XML to CSV conversion phase. The amount of data being processed was so large that reprocessing every file when wanting to update the database would have taken too much time and computing power. So, making the code able to process only the new files and add them to the already existing database would reduce the time and processing power needed to run the program after the first run. Running the data at least once a day required that I implement a scheduler and then run the code once a day.

<u>Understanding the data being processed</u>:

I was working with data from access point that were checking over many variables important to the access point's functionality. The data contains information on how many simultaneous users were active during the window when the data was collected. Inside the files are the Controller level counters and the Node level counters, each with a “P” value corresponding to a specific variable. These P values will be between 1 and 261 for controller level counters, which I put into a separate file of just Controllers. The same goes for Node level counters, but the P values are just greater than 262, which is also put into its own CSV file for Node level counters. Each file represents 15 minutes of collected data and has 1 set of controller counters and N node counters, where N is the number of nodes in that system.

Example of a Controller level counter:

<measValue measObjLdn="EUtranFunction=056852"

Example of a Node level counter:

measObjLdn="EUtranFunction=056852,EUtranCellFDD=056852_1_1">

Example Data:

<measType p="566">SCE_LTE_RRC_Cell.NumUEsWithDRBHist1</measType>

<r p="566">[83957 0 0 0 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 0 0 0 0 0 0 0 10]</r>

The data given is contained in only two ways in the XML file: in counters represented as a singular integer and the other as an array containing multiple integers. To understand the data, it would be read that during this time, there were periods where there were 83957 samples of 0 users, 10 samples of 4 users, 10 samples of 20 users, and 10 samples of 28 users. I decided to take the max and average of each array, ignoring the zero values, to graph these arrays against time. This was the best option because the arrays are not a constant length between files.

<u>Getting set up</u>:

For this project, besides using a regular code editor like VS code, which I used, I needed to install and configure both PostgreSQL, Docker, and Apache-Superset. Starting with PostgreSQL, I began installing the newest downloader found at https://www.postgresql.org/download/ and running the installer once done downloading. Once at the setup menu, click next until you get to the point where you select components to be installed, and once you deselect the stack builder. Select your password and keep the default for everything afterward until you hit finish the install. Once that is installed, you will want to find the Postgresql.conf file, which should be found here but replacing <version> with whatever number version you installed C:\Program Files\PostgreSQL\<version>\data\postgresql.conf. Once there, you will want to ensure inside the file that when it says the listen_addresses, it must equal a '*' like this (listen_addresses = '*'). This will allow PostgreSQL to accept connections from any IP address; you could also set it to any particular IP address or to 'localhost' if local connections are wanted. Make sure to save this file as postgresql.conf. Next, find pg_hbs.conf, which will be in the same directory as postgresql.conf, and add this line of code at the bottom of the file if you want to allow connections from remote IPs:

(host all all 0.0.0.0/0 md5)

Add this line of code if you want to allow connections from local addresses:

(host all all 127.0.0.1/32 md5)

Save this file as pg_hba.conf, go to the services window using Win + R, and type services.msc, right-click on the PostgreSQL service and select Restart. Once this is complete, enter the pgAdmin 4 app, which can control your databases. Once in the app, log into the database using your set password. Right-click on "Databases" and select Create, then Databases. Enter the name you want for the database, and you will want to create two of these, one for nodes and the other for controllers. While in the app, take note of the properties tag when clicking on the "PostgreSQL 17" tag, and remember your hostname/address, port, and username. Next, install Docker Desktop at this link (https://www.docker.com/), run the installer, and use the default setting for the configuration that comes up. To install the Apache Superset using Docker Compose, ensure your Docker Desktop is installed and you have git installed on the computer. To install Git, go to this link (https://git-scm.com/downloads/win) and download the latest 64-bit version of Git for Windows if you have a Windows computer. Run the installer, use the default settings, and install the program. After this, open a command prompt and input these lines of code:

```
git clone --depth=1  https://github.com/apache/superset.git
```

To clone Superset's repo in your terminal and see if this works, you should see a new superset folder in your current directory. Next, enter: `docker --version`" and "docker-compose --version" to check if both are installed. Following that in the command line, enter "mkdir superset-docker", and inside this folder, create a file named "docker-compose.yml" and put this code inside the file:

```yaml
version: "3.1"
services:
  superset:
    image: apache/superset:latest
    container_name: superset
    environment:
      SUPERSET_HOME: /app/superset_home
      SUPERSET_ENV: production
      SUPERSET_CONFIG_PATH: /app/pythonpath/superset_config.py
    volumes:
      - superset_home:/app/superset_home
      - ./superset_config.py:/app/pythonpath/superset_config.py
    ports:
      - "8088:8088"
    depends_on:
      - superset-db
      - superset-cache
    networks:
      - superset-network

  superset-db:
    image: postgres:12
    container_name: superset-db
```

```
    environment:
      POSTGRES_USER: superset
      POSTGRES_PASSWORD: superset
      POSTGRES_DB: superset
    volumes:
      - superset-db-data:/var/lib/postgresql/data
    networks:
      - superset-network

  superset-cache:
    image: redis:latest
    container_name: superset-cache
    networks:
      - superset-network

volumes:
  superset_home:
  superset-db-data:

networks:
  superset-network:
```

Also, create another file inside of the superset-docker folder and create a file named superset_config.py and put this code into the file:

```
SECRET_KEY =
"B6+a5A79Y/bO/A1aLG7T7Bx6F+dxZ9A/5TvBXV6w92K7JbF4g09hWA=="
```

To check if the files successfully made it into the directory, navigate to the directory by typing "cd superset-docker" in the command prompt and then "dir" to see what files are in it.

Put "docker-compose down" in the terminal to stop and remove existing containers. Next enter "docker-compose up --build -d" to rebuild and start the containers, and check the container status with “docker ps -a”. Inside a new terminal, enter “docker exec -it superset /bin/bash” and run the following code to create an admin user and replace the information with your information:

```
superset fab create-admin \
  --username admin \
  --firstname Admin \
  --lastname User \
  --email admin@example.com \
  --password admin
```

Then, initialize the database and Superset with the codes “superset db upgrade” and “superset init.” Use the code “exit” to exit the container. From here, you can go to your browser and go to http://localhost:8088 to access your Superset UI and log in using the admin credentials you created. From this point, after this setup, you should just be able to use your docker desktop to start, stop, and control your containers from their GUI, and you won't need to use the command prompt to access the start docker. To troubleshoot any issues, you can check the logs for errors using “docker logs superset” or try restarting the container “docker restart superset”. Once inside the Apache Superset and you log in, to connect your PostgreSQL data set to superset, go to “Settings” and then “Database Connections”, click the “+ Database” button and choose “PostgreSQL” input the following information of your database and connect to the database. This data should show up when you go to “+Datasets” and choose your desired database. From this point forward, you can make interactive and informative graphs using the data from your database.

Code Details:

Here, I will explain each part of the code, highlight important aspects to notice while implementing it, and discuss any adjustable variables that can be changed in the future to suit a particular need.

Additional Files Needed

In addition to the code files, you will also need an _init_.py file that contains only this line of code "# xml_processor/__init__.py" and save it in your project file. You also need a _pycache_ folder containing compiled Python bytecode files inside your project folder. These files are typically generated by Python when a script is executed, and Python uses them to speed up the loading of modules.

main.py

```
import os
import time
import schedule
import subprocess
from xml_processor.file_handler import get_files
from xml_processor.xml_converter import parse_xml, extract_data,
write_csv, write_combined_csv
```

The "os" library handles file and directory paths; the "time" library tracks execution time and pauses between actions; the "schedule" schedules functions to run periodically; the "subprocess" Runs external Python scripts in new processes. The imports call functions that are found in separate files and are used in the main file.

```
file_path =
'C:/Users/larso/OneDrive/Documents/Project/TestFile/Test1'
controller_output_dir =
'C:/Users/larso/OneDrive/Documents/Project/ControllerOutput'
node_output_dir =
'C:/Users/larso/OneDrive/Documents/Project/NodeOutput'
controller_script =
'C:/Users/larso/OneDrive/Documents/Project/ControllerSQLConv.py'
node_script =
'C:/Users/larso/OneDrive/Documents/Project/NodeSQLConv.py'
```

These are the file paths that are used is “file_path” which is the file that stores all the xml files; “controller_output_dir” is where the converted controller cvs files are stored, while “node_output_dir” is where the converted node cvs files are stored. The “controller_script” and “node_script” are the controller and node codes to insert the CSV data into the PostgreSQL database.

```
run_count = 0
combined_controller_data = []
combined_node_data = []
```

“Run_count” tracks how many times the code has been executed. While “combined_controller_data” and “combined_node_data” store all processed data across multiple runs to be put into a cumulative file

```
def run_additional_scripts():
    controller_process = subprocess.Popen(['python',
controller_script], shell=True)
    node_process = subprocess.Popen(['python', node_script],
shell=True)
    return controller_process, node_process
```

This function runs the external scripts “ControllerSQLConv.py” and “ NodeSQLConv.py” as background subprossing using “subprocess.Popen()”

```
def convert_xml():
    global run_count
    start_time = time.time()
    first_run = (run_count == 0)

    print(f"Run count: {run_count}, First run: {first_run}")
```

This function is responsible for processing xml files, run_count keeps track of how many times the function was executed, and first_run checks if it's the first run of the code.

```
files_to_process = get_files(file_path, first_run)
    # Process each XML file
    for full_file_path in files_to_process:
        print(f"Processing file: {full_file_path}")
        root_element = parse_xml(full_file_path)
        if root_element is not None:
            try:
                # Extract controller and node data from the XML
                ControllerCounter, NodeCounter =
extract_data(root_element)

                # Append the extracted data to the global lists
                combined_controller_data.extend(ControllerCounter)
                combined_node_data.extend(NodeCounter)

                # Generate output file names based on the input file
name
                controller_output_path =
os.path.splitext(os.path.basename(full_file_path))[0] + "_ControlOut"
                node_output_path =
os.path.splitext(os.path.basename(full_file_path))[0] + "_NodeOut"

                # Write individual CSV files for the controller and
node data
                write_csv(controller_output_dir,
controller_output_path, ControllerCounter,
"measObjLdn,,p,max_value,avg_value,beginTime,endTime")
                write_csv(node_output_dir, node_output_path,
NodeCounter, "measObjLdn,p,max_value,avg_value,beginTime,endTime")

            # Handle potential errors that might arise during
processing
            except ValueError as e:
                print(f"ValueError: {e} - File: {full_file_path}")
            except Exception as e:
                print(f"Unexpected error: {e} - File:
{full_file_path}")
        else:
            print(f"Skipping file due to parse error or missing
```

```
element: {full_file_path}")
```

The files_to_process list is populated by calling get_files() with the path to the XML files and a flag indicating whether it's the first run. A loop then iterates over each file, printing the file path and using parse_xml() to parse the XML content. If the XML is parsed without error, extract_data() is called to retrieve ControllerCounter and NodeCounter data, which are then appended to the global combined_controller_data and combined_node_data lists to accumulate data across multiple files. Output file names are generated by removing the extension from the original file name and adding _ControlOut or _NodeOut for controllers and nodes, respectively. These files are then written to their directories using write_csv() with specified headers. If any ValueError or generic exception occurs, the error is logged without stopping the entire process. If the XML cannot be parsed, the file is skipped, and a message is printed to indicate the issue.

```
end_time = time.time()
    total_time = end_time - start_time
    print(f"Total time taken: {total_time:.2f} seconds")

    run_count += 1
```

This part of the code records how long it took to complete, and it increases the run_count counter to keep track of how many times the code has been run.

```
 write_combined_csv(controller_output_dir, combined_controller_data, 
"ControllerCounter_All", 
"measObjLdn,,p,max_value,avg_value,beginTime,endTime")
    write_combined_csv(node_output_dir, combined_node_data, 
"NodeCounter_All", 
"measObjLdn,p,max_value,avg_value,beginTime,endTime")

schedule.every(1).minute.do(convert_xml)

# Run the additional controller and node scripts as background
```

```
processes
controller_process, node_process = run_additional_scripts()
```

The write_combined_csv() function writes the accumulated controller and node data to combined CSV files in their respective directories with the specified headers. The schedule.every(1).minute.do() schedules the convert_xml() function to run every minute. The schedule timing can be adjusted to whatever best fits your data. The run_additional_scripts() function launches the controller and node scripts as background processes using subprocess.Popen().

```
try:
    while True:
        schedule.run_pending()
        time.sleep(1)
except KeyboardInterrupt:
    controller_process.terminate()
    node_process.terminate()
    print("Terminated additional scripts.")
```

This try-except block starts an infinite loop where schedule.run_pending() checks for and executes any scheduled tasks, with a time.sleep(1) to pause the loop for 1 second between iterations, preventing high CPU usage. If the user interrupts the script, the KeyboardInterrupt exception is caught, and both background processes (controller_process and node_process) are terminated using terminate().

xml_converter.py

```
import xml.etree.ElementTree as ET
import os
import csv
import numpy as np
```

“import xml.etree.ElementTree” provides the function to parse and create XML data, “import os” allows for interacting with the operating system for handling file paths and directories. “import csv” is a library that allows for reading from and writing to CSV files and easy handling of tabular data and “import numpy” is a library for doing numerical computations.

```
def parse_xml(file_path):
    try:
        tree = ET.parse(file_path)
        root_element = tree.getroot()
        return root_element
    except ET.ParseError as e:
        print(f"ParseError: {e} - File: {file_path}")
        return None
```

The parse_xml() function attempts to parse an XML file at the specified file_path and returns the root element of the XML tree. If the file is successfully parsed, the root element is returned; otherwise, if a ParseError occurs, an error message is printed, and the function returns None to indicate a failure in parsing the XML.

```
def calculate_avg_ignoring_zeros(data):
    filtered_data = [x for x in data if x != 0]
    if filtered_data:
        return sum(filtered_data) / len(filtered_data)
    else:
        return 0
```

The calculate_avg_ignoring_zeros() function computes the average of a list while ignoring any zero values. It first filters out the zeros from the input data using a list comprehension. If there are non-zero elements in the filtered list, the average is returned by dividing the sum of the filtered data by its length. If the filtered list is empty, it returns 0. This ensures that zero values do not affect the average calculation.

```
def extract_data(root_element):
    beginTime_element =
root_element.find('.//{*}fileHeader/{*}measCollec')
    if beginTime_element is None or 'beginTime' not in
beginTime_element.attrib:
```

```
        raise ValueError("Cannot find beginTime attribute")
    beginTime = beginTime_element.get('beginTime')

    endTime_element =
root_element.find('.//{*}fileFooter/{*}measCollec')
    if endTime_element is None or 'endTime' not in
endTime_element.attrib:
        raise ValueError("Cannot find endTime attribute")
    endTime = endTime_element.get('endTime')

    ControllerCounter = []
    NodeCounter = []
```

The extract_data() function extracts beginTime and endTime from an XML root_element. It searches for the beginTime attribute within the fileHeader/measCollec element and raises a ValueError if the attribute or element is missing. Similarly, it looks for the endTime attribute within fileFooter/measCollec, raising an error if not found. If successful, the function retrieves these values using .get() and initializes two empty lists, ControllerCounter and NodeCounter, for further data extraction.

```
for measValue in root_element.findall('.//{*}measValue'):
        measObjLdn_full = measValue.get('measObjLdn')
        if measObjLdn_full is None:
            raise ValueError("Cannot find measObjLdn attribute")
        measObjLdn = measObjLdn_full.split(',')[1] if ',' in
measObjLdn_full else measObjLdn_full
```

The loop iterates over all measValue elements in the XML root_element using the .findall() method with an XPath query that allows for namespace wildcards (.//{*}measValue). For each measValue element, it retrieves the measObjLdn attribute using .get(). If the attribute is missing, a ValueError is raised. If the attribute is found, the code checks if it contains a comma. If a comma is present, it splits the string and assigns the second part to measObjLdn; otherwise, it assigns the full attribute value to measObjLdn. This ensures proper extraction of the relevant portion of measObjLdn.

```
 for r in measValue.findall('.//{*}r'):
            p = r.get('p')
            values_text = r.text.strip()
            try:
                if '[' in values_text and ']' in values_text:
                    values_list = [int(x) for x in
values_text.strip('[]').split()]
                    max_value = max(values_list)
                    avg_value =
calculate_avg_ignoring_zeros(values_list)
                else:
                    values = int(values_text)
                    max_value = values
                    avg_value = values if values != 0 else 0
            except ValueError:
                raise ValueError(f"Invalid format in measValue:
{values_text}")


            if int(p) <= 261:
                ControllerCounter.append([measObjLdn, "", p,
max_value, avg_value, beginTime, endTime])
            else:
                NodeCounter.append([measObjLdn, p, max_value,
avg_value, beginTime, endTime])

    return ControllerCounter, NodeCounter
```

The code processes each r element within measValue, extracting the p attribute and the text value. If the text contains brackets, it converts it into a list of integers, calculates the maximum value, and computes the average, ignoring zeros. If there's no bracket, it converts the text directly into an integer and assigns that value to both max_value and avg_value, setting avg_value to 0 if the value is zero. If a ValueError occurs during conversion, an exception is raised. Based on the value of p, the data is appended to either ControllerCounter (if p ≤ 261) or NodeCounter (if p > 261). The function returns both lists.

```
def write_csv(output_dir, name, data, headers):
    if data:
        output_path = os.path.join(output_dir, f"{name}.csv")
        with open(output_path, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(headers.split(","))
            for row in data:
                writer.writerow(row)
```

The write_csv() function writes data to a CSV file. It first checks if the data is not empty. If there is data, it creates the file path by joining output_dir and the desired name with a .csv extension. It opens this file in write mode, creates a CSV writer object, and writes the headers (split by commas). Then, it writes the row to the CSV file for each row in the data. This ensures data is saved in the specified CSV format with the given headers.

```
def write_combined_csv(output_dir, combined_data, name, headers):
    if combined_data:
        output_path = os.path.join(output_dir, f"{name}.csv")
        with open(output_path, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(headers.split(","))
            for row in combined_data:
                writer.writerow(row)
```

The write_combined_csv() function writes accumulated combined_data to a CSV file, similar to the write_csv() function. It first checks if combined_data is not empty. If data exists, it constructs the file path using output_dir and the given name, appending a .csv extension. The file is opened in write mode and initializes a CSV writer. The headers are written in the file after being split by commas, and each row in combined_data is written in the file. This function ensures the combined data is saved into a CSV file with the specified headers.

file_handler.py

The libraries used in this code were used in precious code, so an explanation is provided above.

```
def get_files(file_path, first_run):
    files_to_process = []
    for root_dir, dirs, files in os.walk(file_path):
        for name in files:
            full_file_path = os.path.join(root_dir, name)
            if name.endswith('.xml') and
os.path.getsize(full_file_path) > 0:
                file_mod_time = os.path.getmtime(full_file_path)
                if first_run or (time.time() - file_mod_time) <= 24 *
60 * 60:
                    files_to_process.append(full_file_path)
                else:
                    print(f"Skipping old file: {full_file_path}")
            else:
                print(f"Skipping empty or non-XML file:
{full_file_path}")
    return files_to_process
```

The get_files() function scans a directory to process XML files. It uses os.walk() to traverse the directory tree starting from file_path. For each file found, it checks if the file has a .xml extension and is non-empty (i.e., file size > 0). If the file is valid, its modification time is returned. If it's the first run, all XML files are added to files_to_process; otherwise, only files modified within 24 hours are included. Old or non-XML files are skipped, and appropriate messages are printed for each. The function returns a list of XML files to process.

ControllerSQLConv.py

```
import pandas as pd
from sqlalchemy import create_engine
import os
from datetime import datetime, timedelta
import time
import schedule
```

Pandas is used for data manipulation and analysis, particularly with structured data like CSV files or databases. sqlalchemy provides tools for interacting with databases, and create_engine is used to establish a database connection. The os module allows interaction with the operating system, such as managing file paths and directories. datetime and timedelta from the datetime module handle date and time operations, enabling manipulation of date-time information and calculation of time differences. The time module provides time-related functions, such as getting the current time or introducing pauses. The schedule library allows for scheduling tasks to run at specified intervals, enabling automated or periodic task execution.

```
DB_USER = 'postgres'
DB_PASSWORD = 'Shasta42'
DB_HOST = 'localhost'
DB_PORT = '5432'
DB_NAME = 'ControlOut'

csv_directory = 
r'C:\Users\larso\OneDrive\Documents\Project\ControllerOutput'
connection_string = 
f'postgresql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}'
engine = create_engine(connection_string)
```

This code sets up the necessary configurations for connecting to a PostgreSQL database and specifies a directory for storing CSV files. It defines the database credentials, including the username (DB_USER), password (DB_PASSWORD), host (DB_HOST), port (DB_PORT), and database name (DB_NAME). The csv_directory variable stores the path to the folder where CSV files will be accessed or saved. The connection_string is constructed using these credentials, formatted so that PostgreSQL can recognize: 'postgresql://username:password@host:port/dbname'. Using this connection string, the create_engine function from SQLAlchemy establishes a database engine, allowing for interactions with the database, such as reading from or writing to it.

```
try:
    with engine.connect() as connection:
        print("Connection to the PostgreSQL database established 
successfully.")
except Exception as e:
    print(f"Connection failed: {e}")

def preprocess_csv(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()

    with open(file_path, 'w') as file:
        for line in lines:
            line = line.replace(',,', ',')
            file.write(line)
```

Using the engine, this code first attempts to establish a connection to a PostgreSQL database.connect() method. If the connection is successful, a message confirming the connection is printed. If the connection fails, an exception is caught, and a failure message and the error are printed. The preprocess_csv() function is also defined to clean up a CSV file. It takes a file path as input, reads the entire file line by line, and then writes the lines back after replacing instances of double commas (',,') with a single comma (,) so that an empty column is not created in the database. This ensures that any redundant commas in the CSV file are removed, preparing the file for further processing or loading into the database.

```
def process_new_csv_files():
    start_time = time.time()
    now = datetime.now()
    time_threshold = now - timedelta(days=1)

    for filename in os.listdir(csv_directory):
        if filename.endswith('.csv'):
            file_path = os.path.join(csv_directory, filename)
            file_mod_time = 
```

```
datetime.fromtimestamp(os.path.getmtime(file_path))
            if file_mod_time > time_threshold:
                preprocess_csv(file_path)
                df = pd.read_csv(file_path, dtype={'measObjLdn':
str})
                table_name = os.path.splitext(filename)[0]
                df.to_sql(table_name, engine, if_exists='replace',
index=False)
                print(f"Data from {filename} has been successfully
inserted into table '{table_name}'.")
    end_time = time.time()
    duration = end_time - start_time
    print(f"Time taken for this run: {duration:.2f} seconds")
schedule.every(2).minutes.do(process_new_csv_files)
```

The process_new_csv_files() function is designed to process and insert new CSV files into a PostgreSQL database. It starts by recording the current time and setting a time threshold to filter files modified in the last 24 hours. It then loops through all files in the specified csv_directory, and for each file with a .csv extension, it checks whether the file's modification time is more recent than the threshold. If so, the file is preprocessed by the preprocess_csv() function to clean up any redundant commas. The cleaned CSV is loaded into a pandas DataFrame, with the measObjLdn column specified as a string. The data is then inserted into a PostgreSQL table, with the table name derived from the CSV file name (minus the extension), using the df.to_sql() function. The table is replaced if it already exists. The function also records the time taken for the entire process and prints this duration. Finally, the function is scheduled to run every 2 minutes using the schedule.every(2).minutes.do() method, ensuring regular, automated processing of newly modified CSV files. The timing variables can be changed to fit the needs of whatever is desired during the task.

```
while True:
    schedule.run_pending()
```

```
    time.sleep(2)
```

This code creates an infinite loop that continuously checks for scheduled tasks using schedule.run_pending(). It will execute any due tasks based on the defined schedule. The time.sleep(2) function pauses the loop for 2 seconds after each iteration to reduce CPU usage and prevent the loop from running continuously without pause. This ensures the scheduled tasks run at their designated intervals, allowing the program to remain responsive.

NodeSQLConv.py

```
import pandas as pd
from sqlalchemy import create_engine
import os
from datetime import datetime, timedelta
import schedule
import time

DB_USER = 'postgres'
DB_PASSWORD = 'Shasta42'
DB_HOST = 'localhost'
DB_PORT = '5432'
DB_NAME = 'NodeOut'

csv_directory =
r'C:\Users\larso\OneDrive\Documents\Project\NodeOutput'
connection_string =
f'postgresql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}'
engine = create_engine(connection_string)
```

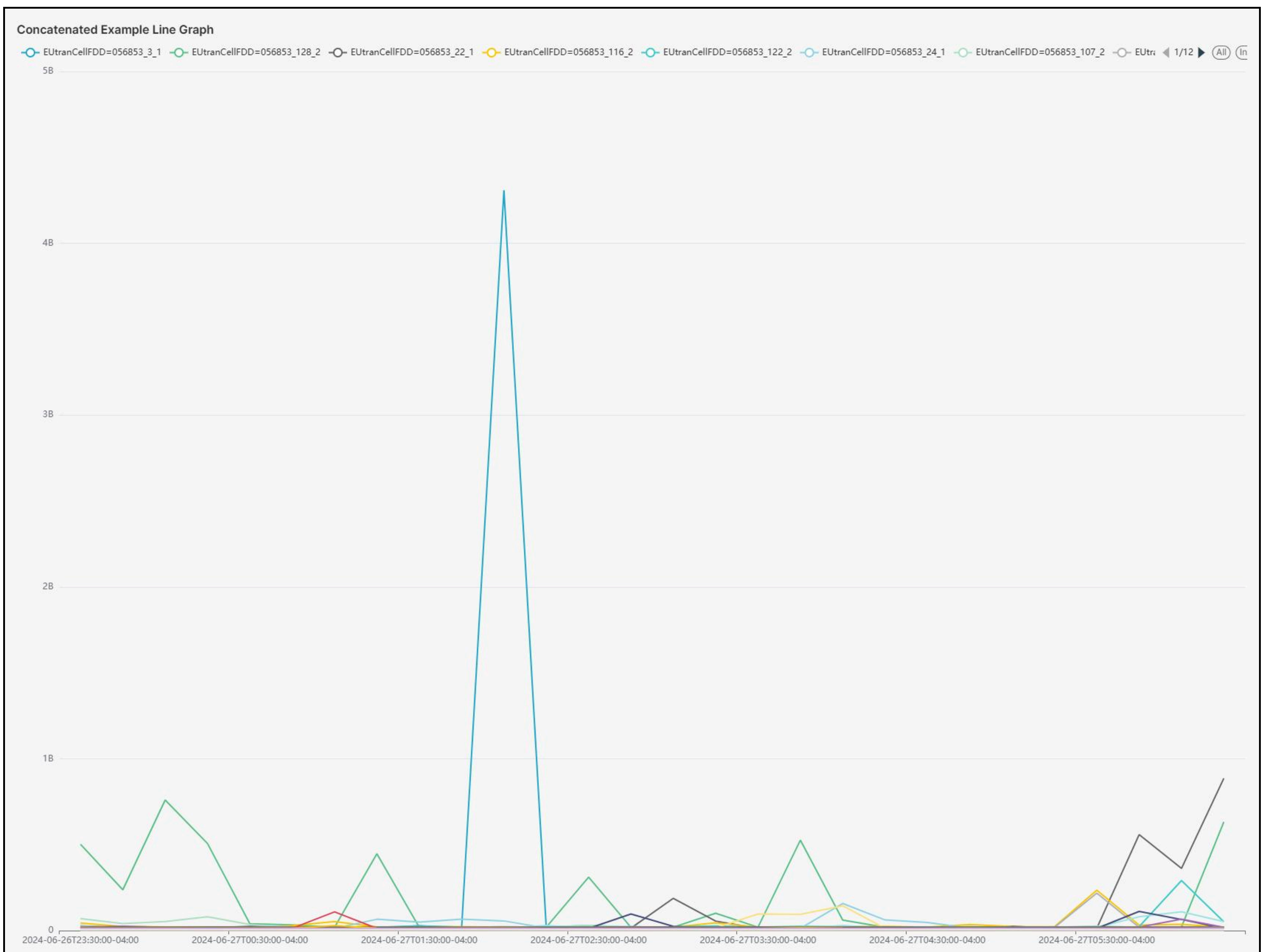
The libraries used in this code section are the same as in the previous code, so explanations of what each library does will be above the previous one. The connection string is the same except for the DB_NAME, which is changed to the node database instead of the controller database. The same goes for the csv_direcotry, which is also switched to the cvs node file.

```
try:
```

```
    with engine.connect() as connection:
        print("Connection to the PostgreSQL database established 
successfully.")
except Exception as e:
    print(f"Connection failed: {e}")

def process_new_csv_files():
    start_time = time.time()
    now = datetime.now()
    time_threshold = now - timedelta(days=1)

    for filename in os.listdir(csv_directory):
        if filename.endswith('.csv'):
            file_path = os.path.join(csv_directory, filename)
            file_mod_time = 
datetime.fromtimestamp(os.path.getmtime(file_path))
            if file_mod_time > time_threshold:
                df = pd.read_csv(file_path)
                table_name = os.path.splitext(filename)[0]
                df.to_sql(table_name, engine, if_exists='replace', 
index=False)
                print(f"Data from {filename} has been successfully 
inserted into table '{table_name}'.")
    end_time = time.time()
    duration = end_time - start_time
    print(f"Time taken for this run: {duration:.2f} seconds")
schedule.every(2).minutes.do(process_new_csv_files)

while True:
    schedule.run_pending()
    time.sleep(2)
```

This code first attempts to connect to a PostgreSQL database using SQLAlchemy's engine.connect(). If successful, it prints a confirmation message; otherwise, an error message will be displayed if the connection fails. The function process_new_csv_files() is then defined to handle the processing of new CSV files. It begins by capturing the current time and establishing a time threshold of the last 24 hours (timedelta(days=1)). The function loops through all files in the specified directory. For each .csv file, the modification time of the file is checked. If the file was modified within the last 24 hours, the CSV is loaded into a pandas DataFrame using pd.read_csv(). The DataFrame is then inserted into a PostgreSQL table whose name is derived from the filename, using the df.to_sql() function, which replaces any existing table with the same name. Once the process is complete, the time taken for the entire run is printed. This function is scheduled to run every 2 minutes using the schedule.every(2).minutes.do(process_new_csv_files) method. Finally, an infinite loop (while True) runs, checking for scheduled tasks using schedule.run_pending(). The loop pauses for 2 seconds after each iteration with time.sleep(2), ensuring the scheduled tasks are executed at their designated intervals without overloading the system.

<u>Example Data Visualization Using Apache Superset</u>:



<u>Challenges and Solutions</u>:

One of the major challenges faced during this project was the complexity and inconsistency of the data. The XML files contained various data structures, such as arrays of varying lengths and singular values, making the data difficult to interpret and standardize. There was no uniform format across the files, and the same variables could appear in different formats, requiring careful parsing and preprocessing. To address this, I developed custom scripts to handle these inconsistencies by extracting key elements, calculating meaningful metrics like maximum and average values, and creating a standardized CSV output. This allowed for more reliable data storage and analysis despite the non-standard nature of the raw XML files. Another significant challenge was setting up the PostgreSQL database to handle incoming data. Firewall restrictions and issues with port listening initially prevented the database from accepting connections, especially when configuring Docker containers. The PostgreSQL database needed to be properly configured to allow external connections, and this required modifications to both the PostgreSQL configuration files and the firewall settings. By updating the postgresql.conf and pg_hba.conf files to enable listening on all available IP addresses and configuring the firewall to allow traffic through the necessary ports. I resolved these connectivity issues. This solution ensured the database could communicate with external services (like Docker and Apache Superset) and process data as intended. These combined solutions allowed for a smooth data pipeline integration, overcoming the challenges of complex data formats and infrastructure-level connectivity issues.

<u>Conclusion and Future Improvements</u>:

This project successfully automated the processing of XML data from an access point into a structured PostgreSQL database, enabling efficient analysis and visualization using Apache Superset. The system effectively handles large datasets, identifies new files for processing, and provides robust error handling, ensuring data integrity despite potential issues such as malformed XML files. By leveraging Docker for containerization, the setup remains scalable and easy to replicate across different environments. Overall, the pipeline achieves the goal of automating data transformation and visualization, making it a valuable tool for continuous data monitoring and analysis. Several improvements could be implemented to enhance the system's performance and data interpretation. First, parallel processing could be introduced to significantly reduce the time required to handle large datasets, especially in environments with many nodes.

Additionally, implementing a more sophisticated method for validating and cleaning data could allow minor errors in the XML files to be corrected rather than skipped. A key area for future improvement is in the graphing of array data. The current approach uses max and average values to represent the data, which works but may need to be more accurate in the underlying patterns. A more advanced graphing method, such as using histograms or time-series plots that visualize the distribution of values across different time windows, could more accurately capture the nuances in the data and provide deeper insights into the access point performance. Furthermore, integrating machine learning models for anomaly detection could automatically highlight unusual patterns, and transitioning to a cloud-based infrastructure would improve scalability, enabling the pipeline to handle even larger datasets efficiently. These future enhancements would make the system more robust, insightful, and adaptable to growing data needs.