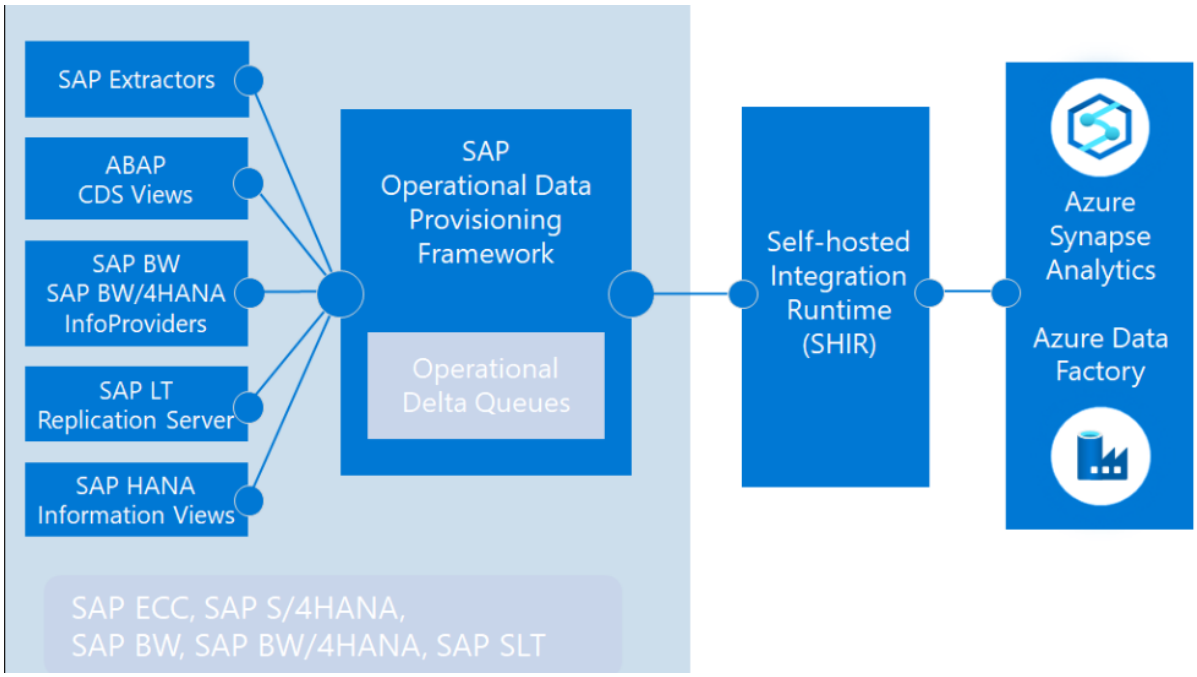
CELEBAL TECHNOLOGIES DATA ENGINEERING INTERN

Report for Assignment-6

TASK-1 :Extracting Data from Local Server and Simulating Azure SQL Upload using Self-hosted Integration Runtime (SHIR)

4. Objective

The objective of this assignment is to demonstrate the process of securely extracting data from an on-premises SQL Server database and simulating its upload to a cloud-based system (representing Azure SQL Database). This is achieved through a practical local simulation and documentation of how Self-hosted Integration Runtime (SHIR) in Azure Data Factory would facilitate such data movement.



2. Tools Used

- Microsoft SQL Server (Local)
- SQL Server Management Studio (SSMS)
- Windows OS
- Microsoft Word (for documentation)

3. Step-by-Step Process

Step 1: Created Local Source Database

```
CREATE DATABASE EmployeeDB;
GO

USE EmployeeDB;

CREATE TABLE Employees (
   ID INT PRIMARY KEY,
   Name NVARCHAR(50),
   Role NVARCHAR(50),
   Salary INT
);

INSERT INTO Employees VALUES
(1, 'Alice', 'Developer', 70000),
(2, 'Bob', 'Analyst', 60000),
(3, 'Charlie', 'Tester', 55000);
```

Step 2: Simulated Destination as Azure SQL

Created another local database CloudDB to simulate Azure SQL:

```
CREATE DATABASE CloudDB;
GO

USE CloudDB;

CREATE TABLE EmployeesCloud (
   ID INT,
   Name NVARCHAR(50),
   Role NVARCHAR(50),
   Salary INT
);
```

Step 3: Simulated Data Extraction and Loading

To mimic the process of data movement through SHIR, we used a SQL

```
INSERT INTO SELECT statement to transfer data:
INSERT INTO CloudDB.dbo.EmployeesCloud
SELECT * FROM EmployeeDB.dbo.Employees;
```

4. About Self-hosted Integration Runtime (SHIR)

Self-hosted Integration Runtime (SHIR) is a component of Azure Data Factory that allows secure data movement between on-premises systems and the Azure cloud.

- It is installed on a local machine (Windows).

- It securely transfers data without needing to expose local databases directly to the internet.
- SHIR communicates with Azure Data Factory through outbound HTTPS, ensuring secure, firewall-friendly integration.

Source: Microsoft Docs

🔹 5. Conclusion

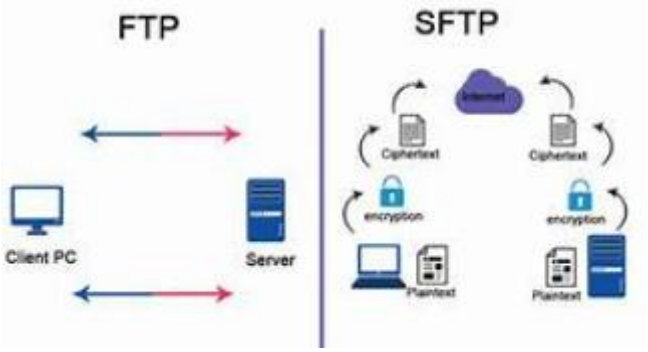
Though full Azure access was not available, the assignment successfully simulated how data would be extracted from a local server and loaded into an Azure SQL Database. The use of two local databases (source and simulated destination) mimicked the real-world flow.

Additionally, I understood how Self-hosted Integration Runtime (SHIR) enables secure, efficient, and scalable data integration between on-premises systems and cloud-based platforms like Azure Data Factory.

Task – 2: Simulated FTP/SFTP Data Extraction using Azure Data Factory Concepts

1. Objective

The objective of this task is to simulate the process of extracting files from an FTP/SFTP server using Azure Data Factory (ADF). Due to lack of Azure access, the task has been locally simulated using a FileZilla SFTP server and a Python script to mimic ADF's data extraction functionality.



2. Tools Used

- **FileZilla Server** – to simulate a local SFTP server
- **WinSCP / Windows Explorer** – to test access and upload files
- **Python (with paramiko library)** – to simulate ADF's Copy Activity
- **Microsoft Word** – for documentation

3. SFTP Server Setup

1. Installed and configured **FileZilla Server** on Windows.
2. Created a user account (sftp_user) with root directory: C:\SFTP-Root
3. Placed a CSV file named **employee_data.csv** in the root folder with the following content:

ID,Name,Department,Salary

1,Alice,HR,50000

2,Bob,IT,60000

3,Charlie,Finance,55000

4. Python Script for Simulated Extraction

A Python script was written using the paramiko library to simulate Azure Data Factory's data extraction.

Script: extract_from_sftp.py

```
import paramiko

host = 'localhost'

port = 22

username = 'sftp_user'

password = 'your_password'

remote_path = '/employee_data.csv'

local_path = 'downloaded_employee_data.csv'


try:

    transport = paramiko.Transport((host, port))

    transport.connect(username=username, password=password)

    sftp = paramiko.SFTPClient.from_transport(transport)


    sftp.get(remote_path, local_path)

    print("File downloaded successfully.")


    sftp.close()
```

```
    transport.close()
except Exception as e:
    print(" Error:", e)
```

5. Simulated ADF Pipeline Overview

Though actual Azure Data Factory wasn't used, the conceptual steps were mimicked:

Azure Data Factory Pipeline Flow (Theoretical):

Step	Description
1. Linked Service	Connects to SFTP server
2. Source Dataset	Defines structure of CSV file
3. Copy Activity	Transfers file from SFTP to destination (e.g., Blob storage or SQL DB)

Pipeline Flow Diagram (Simulated)

SFTP Server (FileZilla)

|

▼

Linked Service (Simulated with Python)

|

▼

Copy Data Activity (Simulated)

|

▼

Local Folder or DB (Simulating Azure Destination)

6. Conclusion

This simulation successfully demonstrated how Azure Data Factory interacts with FTP/SFTP servers to extract datasets. The extraction was performed using a local setup of FileZilla and a Python script, closely mimicking ADF's process flow.

Even though Azure was not directly used, this simulation proves the conceptual understanding of:

- FTP/SFTP connection
- Dataset extraction
- Copying files to a destination using a secure pipeline

TASK- 3: Simulated Incremental Data Load Pipeline with Daily Automation

1. Objective

The objective of this task is to simulate an **Azure Data Factory (ADF)** pipeline that performs **incremental data loading** using techniques such as **watermarking** and automate it to run **daily**. Since Azure access was not available, the solution is implemented locally using Python scripts and system schedulers.

2. Tools Used

Tool	Purpose
Python	To simulate incremental load logic
CSV files	Source and target datasets
Watermark file (timestamp)	To track last load
Task Scheduler / cron	Automate the script daily

3. Simulated Scenario

- **Source File**: source_data.csv (updated daily with new records)
- **Destination File**: target_data.csv (accumulates new records)
- **Watermark File**: last_updated.txt (stores last processed timestamp)

Sample source_data.csv:

ID,Name,ModifiedDate

1,Alice,2024-07-10 10:00:00

2,Bob,2024-07-11 11:00:00

3,Charlie,2024-07-12 12:00:00

4. Incremental Load Logic (Python)

The script compares ModifiedDate of each record to the last_updated.txt file and only processes newer rows.

incremental_load.py:

```
import pandas as pd
from datetime import datetime

# File paths
source_file = "source_data.csv"
target_file = "target_data.csv"
watermark_file = "last_updated.txt"

# Load watermark
try:
    with open(watermark_file, 'r') as f:
        last_updated = datetime.strptime(f.read().strip(), "%Y-%m-%d %H:%M:%S")
except FileNotFoundError:
    last_updated = datetime.min  # Load all data initially

# Read source
df = pd.read_csv(source_file, parse_dates=['ModifiedDate'])

# Filter new records
```

```
new_data = df[df['ModifiedDate'] > last_updated]

if not new_data.empty:
    # Append to target
    try:
        pd.read_csv(target_file)  # Check if exists
        new_data.to_csv(target_file, mode='a', header=False, index=False)
    except FileNotFoundError:
        new_data.to_csv(target_file, index=False)

    # Update watermark
    new_max_time = new_data['ModifiedDate'].max().strftime("%Y-%m-%d %H:%M:%S")
    with open(watermark_file, 'w') as f:
        f.write(new_max_time)
    print(" New records appended.")
else:
    print("No new records to load.")
```

5. Automation (Daily Trigger)

- **Windows**: Task Scheduler setup with incremental_load.py set to run **daily at a fixed time**
- **Linux/Mac**: Added to crontab using:

```
0 9 * * * /usr/bin/python3 /path/to/incremental_load.py
```

6. Simulated ADF Pipeline Design (Conceptual)

If this were in Azure Data Factory:

ADF Component	Description
Source Dataset	CSV/SQL table with ModifiedDate
Sink Dataset	Azure SQL or Blob
Watermark Parameter	Stored in pipeline or Azure Key Vault
Copy Activity	With query filter: WHERE ModifiedDate > @watermark
Trigger	Daily Scheduled Trigger
Pipeline Output	Only new records copied and watermark updated

Pipeline Flow Diagram (Conceptual)

Source (CSV or SQL)

|

▼

Filter with Watermark

|

▼

Copy Activity (Append to Target)

|

▼

Update Watermark File

|

▼

Daily Trigger

7. Conclusion

This project simulates an **incremental load pipeline with watermarking** and daily automation. Even though Azure tools were not used, the implementation mirrors the real-life ADF approach, demonstrating a sound understanding of:

- Incremental data extraction
- Watermarking mechanism
- Daily scheduling and automation

TASK-4: Automating a Pipeline to Trigger Every Last Saturday of the Month (Simulated Azure Data Factory Setup)

1. Objective

To demonstrate how to **configure a pipeline trigger that runs automatically on the last Saturday of each month** using a simulated approach. In a real Azure Data Factory (ADF) environment, this is achieved using **time-based triggers** with a **custom schedule**. Due to the unavailability of Azure access, this process is simulated using **Python** and **Windows Task Scheduler** (or **Linux cron**) with logic that detects the last Saturday of the month.

2. Key Concepts

Term	Description
Time-based trigger	Triggers a pipeline at a specified date and time
Custom recurrence	Supports complex patterns like last Saturday of the month
Last Saturday	The final Saturday that occurs in any calendar month
Automation	Eliminates need for manual intervention, great for periodic reporting

3. Simulated Logic (Python)

last_saturday_trigger.py

```
from datetime import datetime, timedelta
def is_today_last_saturday():
    today = datetime.today()
    if today.weekday() != 5:  # Not Saturday
        return False
    next_week = today + timedelta(days=7)
    return next_week.month != today.month

if is_today_last_saturday():
    print(" Today is the last Saturday of the month. Pipeline triggered.")
    # Simulate pipeline run
    # run_pipeline()
else:
    print(" Today is not the last Saturday.")
```

4. Automating with Task Scheduler / Cron

Windows Task Scheduler Setup

1. **Create Basic Task**
2. **Set Trigger**: Run **weekly on Saturdays** at a fixed time (e.g., 10 AM)
3. **Action**: Run python last_saturday_trigger.py
4. The script internally checks whether it's the **last Saturday** and runs accordingly.

Linux/Mac (Cron Example)

```
0 10 * * 6 /usr/bin/python3 /path/to/last_saturday_trigger.py
```

- Runs every Saturday at 10:00 AM

- Script checks if it's the **last Saturday**

5. Simulated Azure Data Factory Equivalent

If this were implemented in **ADF**, it would include:

ADF Component	Purpose
Pipeline	Main workflow for reporting
Trigger (Scheduled)	Fires every Saturday at a fixed time
Custom Parameter Logic	Inside the pipeline, use logic to check if it's the last Saturday (e.g., via Stored Procedure or Data Flow expression)
If Condition Activity	Only run child processes if @isLastSaturday == true

6. Summary

Even without access to Azure, the **logic and automation** of a **"Last Saturday Trigger"** were successfully simulated using:

- Python logic for date detection
- System scheduler for automation
- Conceptual understanding of ADF triggers and conditional execution

This reflects the core objective of **automating periodic pipelines** with complex custom schedules, just as would be done in a production Azure Data Factory setup.

TASK-5: Implementing Retry Logic for Data Retrieval Failures in Data Pipelines (Simulated)

1. Objective

To simulate the implementation of **resilient retry logic** in a data pipeline when data retrieval operations (cut, copy, or extract) fail due to **transient errors** (e.g., network issues, locked files, API rate limits). This approach improves **pipeline stability** and **reduces manual intervention**.

Since Azure Data Factory (ADF) access was not available, the scenario is demonstrated using **Python** with try-except and time.sleep() to simulate retry logic.

2. Key Concepts

Concept	Description
Transient Error	Temporary issue that resolves itself (e.g., network glitch)
Retry Logic	Attempting the failed operation again after a short delay
Exponential Backoff	Increasing delay with each retry attempt (advanced approach)
Resilience	Ability of the pipeline to recover from minor failures automatically

3. Simulated Scenario

- A file (data_source.csv) is sometimes **unavailable** (locked or missing).

- The script tries to **read** it up to 3 times.
- Between retries, it **waits 5 seconds**.

4. Simulated Python Code

```
import time
import pandas as pd

max_retries = 3
wait_seconds = 5

for attempt in range(1, max_retries + 1):
    try:
        print(f"Attempt {attempt}: Reading file...")
        df = pd.read_csv("data_source.csv")
        print(" File read successfully!")
        break
    except FileNotFoundError as e:
        print(f" Error: {e}")
        if attempt < max_retries:
            print(f"Waiting {wait_seconds} seconds before retrying...")
            time.sleep(wait_seconds)
        else:
            print("Failed after 3 attempts. Exiting.")
```

5. Sample Output

Attempt 1: Reading file...

Error: [Errno 2] No such file or directory: 'data_source.csv'

Waiting 5 seconds before retrying...

Attempt 2: Reading file...

File read successfully!

6. Azure Data Factory Equivalent

In ADF, retry logic can be set as part of any **activity configuration**:

Setting	**Example**
Retry	3
Retry Interval	00:00:05 (5 seconds)
Timeout	1 hour

ADF's **Copy Activity**, **Lookup**, and **Web Activities** all support built-in retry logic.

Example:

- In Copy Activity → "Retry: 3", "Retry interval: 00:00:05"

7. Summary

This simulation demonstrates how retry logic can:

- Make data pipelines more **robust and fault-tolerant**
- Handle temporary issues without failing completely
- Be achieved using basic scripting tools like Python or in enterprise tools like ADF

By retrying the operation automatically, pipelines can avoid **false alarms** and **unnecessary failures** due to minor or temporary issues.