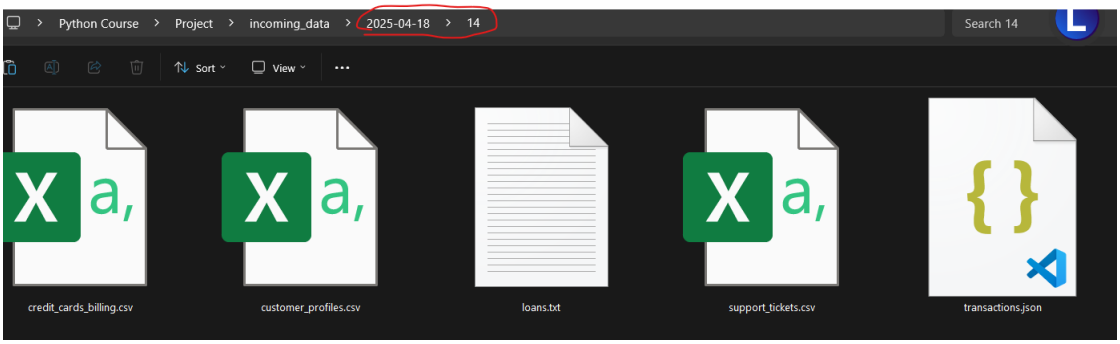
DISCLAIMER: This data was generated automatically using a Python script and does not reflect any real-world information.

Project Description

- You have been hired by **NexaBank**, a **Banking company** experiencing high **customer churn**.
- The company plans to develop a **machine learning** model to predict customers who are likely to switch to one of the competitors.
- These at-risk customers will be targeted by the marketing team with special offers and retention strategies.
- As the **Data Engineer**, your role is to **build a clean, well-engineered dataset** by **integrating and transforming messy, multi-source customer data**, making it ready for churn prediction models.

Data Flow

- The data is sourced from multiple systems and arrives in various file formats, including **JSON**, **CSV**, and **TXT**.
- We receive the data in **real-time**, meaning it is sent to us immediately after extraction from the source systems.
- Each file follows a **static, predefined schema**.
- The data is organized using a **two-level partitioning** structure (**date** and **hour**) as follows:



Data for 2025-04-18 2PM

- For example, data for **April 18th, 2025,** is stored in a folder named **2025-04-18**, containing **24 subfolders**, one for each hour of the day.
- Occasionally, the source sends:
  - Corrupted or invalid data files.
  - Files with incorrect schemas.
  - Extra/unnecessary data files.

Data Sources

You have to combine **multiple messy sources** of different formats:

1. **Customer Profile - CSV (customer_profiles.csv)**
   This file contains data about our customer base.
   - customer_id, name, gender, age, city, account_open_date, product_type, customer_tier.

2. **Credit Cards Billing - CSV (credit_cards_billing.csv)**
   This file contains data on credit cards monthly payment for credit card customers.
   - bill_id, customer_id, month, amount_due, amount_paid, payment_date.

3. **Support Tickets - CSV (support_tickets.csv)**
   This file contains data related to customer complaints.
   - ticket_id, customer_id, complaint_category, complaint_date, severity.

4. **Loans - TEXT (loans.txt)**
   This file contains data on loans requested by customers.
   - customer_id, loan_type, amount_utilized, utilization_date, loan_reason.

5. **Money Transfers & Purchases - JSON (transactions.json)**
   This file contains data on money spent by bank customers.
   - sender, receiver, transaction_amount, transaction_date.

- The data comes from sources almost clean and of high quality.
- So, no heavy data cleaning or quality checks required.

Requirements

- Data is received from sources inside a directory named **"Incoming_data"**.
- Your task is to **loop through all files** in this directory and **apply the specified transformations** to each file based on its type.
- For customers data:
  - Add a new **Integer** column named **tenure**, representing the number of years since the customer joined the company.
  - Add a new **String** column named **customer_segment**, based on the following criteria:
    - **Loyal:** if the customer has been with the company for more than 5 years.
    - **Newcomer:** if the customer has been with the company for less than 1 year.
    - **Normal**: otherwise.
- For credit card billing data:
  - Add a new **Boolean** column named **fully_paid**, set to **True** if the customer has paid the full bill amount, and **False** otherwise.
  - Add a new **Integer** column named **debt**, representing the remaining due amount after payment.
  - Add a new **Integer** column named **late_days**, representing the number of days between the bill's **due date** (always the **1st of each month**) and the **actual payment date**.
  - Add a new **Float** column named **fine**, representing the fine charged to customers for late payments, calculated as:
    - **Fine = late_days * 5.15**
  - Add a new **Float** column named **total_amount**, calculated as:
    - **total_amount = amount_due + fine**
- For support tickets data:

  Add a new **Integer** column named **age**, representing the number of days since the ticket was issued.
- For transactions data:
  - Knowing that each transaction costs the customer **50 cents + .1 %** of the transaction amount.
  - Add a new **float** column named **cost**, representing the cost of the transaction.
  - Add a new **float** column named **total_amount**, representing **transaction_amount + cost**.
- For loans data:
  - Add a new **Integer** column named **age**, representing the number of days since the transaction was completed.
  - If the **loan** costs the customer **20%** of its value per year, in addition to **1K$**.
  - Add a new **float** column named **total_cost**, representing **annual cost for the loan.**

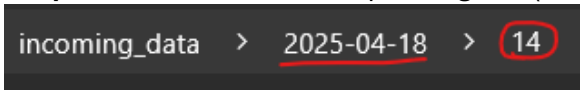
For loan reason data:

- This column contains **sensitive** information (**the reason for the loan request**).
- The **Data Science team** must **not** have access to the **plain text** of the loan reason!
- Your task is to **encrypt each row** using a **Caesar cipher**.
- **Caesar cipher** encryption/decryption depends on a **key**.
- **Important:** You **cannot** use a **static key** for **encryption**!
- You must **generate a random key** for encrypting each file.

Hints and guidelines:

- You can maintain a **state store** with **two columns**:
  - **File name**
  - **Encryption Key**
- Alternatively, you can use a **keyless decryption mechanism**:
  - Encrypt the **text** using a **random key without storing** the key.
  - To decrypt:
  - You will be provided with a **text file listing all valid English words**!
  - Write a function that checks a sentence and **counts how many valid English words** it contains.
  - Try to decrypt the message using all possible **Caesar cipher shifts** (keys from 1 to 25).
  - For each trial count the number of **valid English words** of the **text's** content.
  - Select the decryption that **yields the highest number of valid English words** — this is considered the correct decrypted text.

- Finally:
  - Add the following **data quality columns** to each table:
    - **processing_time**: the current timestamp at the time of processing.
    - **partition_date**: the date part, e.g., **2025-04-18** (based on the example provided).
    - **partition_hour**: the hour part, e.g., **14** (based on the example provided).



  - Drive meaningful insights from the transformed data:
    - Perform an **analysis** of your choice.
    - Focus on delivering **insightful** findings.
    - **Creativity** is highly encouraged in your **analysis**.
  - Identify **at least three possible reasons** behind the company's **high customer churn rate** based on your findings.
  - Save the final output:
    - Save the 5 processed tables in **Parquet format**.
    - Upload the files to **HDFS** Using Python **subprocess**.
    - These files will be inserted into **Hive tables** for the Data Science team to access.

Instructions

Logging Requirements:

- **Every step** of the pipeline must be properly **logged**:
  - Each log entry must include **Timestamp**, **Action**, and **Status:**
    - **Ex:** 2025-04-19 01:17:38 => Started extracting loans data from txt file at "C:\Users\alghaly\Desktop\Python Course\Project\incoming_data\2025-04-18\14\loans.txt"
  - Logs should capture **as many details as possible**, such as:
    - Number of rows extracted.
    - Extracted schema.
    - Any other useful metadata or operational details.
  - **Persist logs** into **files** for future **log analysis**.

Pipeline Behavior:

- The pipeline must operate in **real-time**.
- Data should be **extracted**, **transformed**, and **loaded immediately** upon receipt.

Schema Validation:

- **Reject** any incoming files that do **not** match the **predefined schema**.
- Upon rejection:
  - **Log** the action with appropriate details.
  - **Send an email notification** to enable **faster issue recovery**.

Error Handling:

- In case of **pipeline failure**:
  - **Log** the exception details **clearly**.
  - **Send an email notification** about the failure.
  - **Automatically re-run** the pipeline.

Real-time Constraints:

- Because the pipeline is **real-time**, **quick failure detection and recovery** are **critical**.
- Again, the pipeline works in **real-time fashion,** so make sure **once** a file is extracted, transformed, and loaded, it **won't be reprocessed** in the next cycle!
  - **Hint:** consider using a state store or an archiving mechanism.

Sensitive Information:

- You will need your **email password** to send automated emails.
- Store your password in a **separate text file** and read it securely within your script.
- **Do not** hardcode or include your password in the project delivery.

Suggested analysis — Optional

Churn Analysis:

- Identify which customer **segments** (by age, location, spending level) have the **highest churn rate**.
- Analyze **churn** correlation with **late credit card payment** behavior.

Behavior Trend Analysis:

- Compare **customer activity** (transactions, loans requests) between churned and non-churned customers.
- Identify **spending patterns** that predict **churn**.

Revenue Analysis:

- Calculate **Average Revenue** Per User (ARPU) and identify the customer profile of **high spenders** vs **churners**.

Geographical Analysis:

- Find **cities** or **regions** with the highest churn rates.

Tenure Analysis:

- Analyze if **newly joined** customers **churn faster** than older ones.

Bonus Section (Optional — No Implementation Required)

You are highly recommended to think about these steps (no implementation needed)

- **Consider replacing** the full data load with an **incremental load** strategy to improve efficiency and performance. *(Interview Question)*
- **Think about implementing** a **log analyzer script** to extract insights from the generated log files.
  - You may implement it using **Python** or **Bash**, whichever you find more convenient.
- **Generate a final report** summarizing:
  - Any **anomalies detected** during the data extraction and transformation processes.
  - This report would assist in **further analysis** and quality checks.
- **Extra Enhancements:**
  - Some of the functionalities like **mailing** don't need to **block the pipeline**.
  - Consider moving these **functionalities** to separate **threads**.
  - The pipeline is **real-time** and very **critical** you are very encouraged to create a separate **thread** for **health checks**.

Delivery

- The **deadline** for the project is **Sunday** 11th **of May (3+ weeks)**.
- The project is **team-based**; you will be divided into **teams of 2s**.
- Students will submit their work via a **sheet containing links to their GitHub repositories**.
- Students are expected to **understand and explain the computational complexity (performance analysis)** of their written code.
- Students are **required** to follow the **Object-Oriented Programming (OOP) paradigm** throughout the project.
- **Applying SOLID principles** is strongly recommended, as it will **make your development process much easier and more organized**.

Good luck, enjoy 😀.