Method of Procedure (MoP) to implement the "End-to-End Telecom Pipeline" in Python + SQLite.

You will:

- Load raw usage data from telecom_raw.csv (e.g. extracting from data lake)
- Clean & transform it, compute total usage per customer and flag heavy-user (ETL pipeline)
    - Aggregate total usage per customer
    - Flag heavy users (>50GB OR >500 minutes)
- Load the cleaned data into a SQLite "mini data-warehouse" with an index.
- Create role-based views for junior analysts vs managers.
- Automate the pipeline to run for 20 seconds for 3 times, with simple logging.

1. Environment setup

Open Anaconda & Jupyter -> Start Anaconda Navigator -> Click Jupyter Notebook.
In the browser tab, create a new folder on the Desktop/telecom_pipeline.
Click new notebook pipeline.ipynb.

Install required libraries (one-time)

```
pip install pandas schedule
```

2. Prepare raw data file (telecom_raw.csv)

For class practice we'll create a small dummy CSV directly from Python.

```
import pandas as pd
raw_data = [
    # customer_id, call_minutes, data_gb,   timestamp,          region
    [1001,        120,         10.5,     "2025-09-25 10:30",  "Delhi"],
    [1002,        450,         55.0,     "2025-09-25 11:00",  "Mumbai"],
    [1003,        200,         20.0,     "25-09-2025 09:15",  "Chennai"],  # different date format
    [1004,        600,         70.0,     "2025/09/25 08:00",  "Delhi"],    # different date format
    [1005,        50,          5.0,      "2025-09-25 12:00",  "Kolkata"],
    [1005,        50,          5.0,      "2025-09-25 12:00",  "Kolkata"],  # duplicate
    [1006,        None,        60.0,     "2025-09-25 13:00",  "Hyderabad"],# missing call_minutes
    [1007,        510,         None,     "2025-09-25 14:00",  "Delhi"],    # missing data_gb
]

columns = ["customer_id", "call_minutes", "data_gb", "timestamp", "region"]
df_raw = pd.DataFrame(raw_data, columns=columns)

df_raw.to_csv("telecom_raw.csv", index=False)
print("Saved telecom_raw.csv")
df_raw
```

3. Common imports and database file

```
import sqlite3
import logging
import schedule
import time
```

```
import os
import pandas as pd
#Set database path and configure logging

DB_PATH = "telecom_warehouse.db"

# Basic logging configuration
logging.basicConfig(
    filename="telecom_pipeline.log",
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
)

print("Database path:", DB_PATH)
```

4. ETL Step - Extract, Transform & Clean

 # Extract

```
def extract_raw_data(csv_path="telecom_raw.csv"):
    """
    Step 1: Extract raw data from CSV file.
    Returns a pandas DataFrame.
    """
    if not os.path.exists(csv_path):
        raise FileNotFoundError(f"{csv_path} not found in current folder.")

    df = pd.read_csv(csv_path)
    logging.info("Extracted %d rows from %s", len(df), csv_path)
    return df
```

 # Function to transform & clean

```
import pandas as pd
import logging

# Function to transform & clean
def transform_and_clean(df_raw):
    """
    Step 2: Clean and transform the data.
    - Remove exact duplicate rows
    - Standardize date format
    - Handle missing values (impute)
    - Aggregate total usage per customer
    - Flag heavy users (>50GB OR >500 minutes)
    """
    df = df_raw.copy()

    # 1) Remove duplicates
    before = len(df)
    df = df.drop_duplicates()
    logging.info("Removed %d duplicate rows", before - len(df))
```

```python
    # 2) Standardize timestamp to datetime
    df["timestamp"] = pd.to_datetime(df["timestamp"], errors="coerce")
    bad_dates = df["timestamp"].isna().sum()
    if bad_dates > 0:
        logging.warning("Found %d bad timestamps; filling with default date", bad_dates)
        df["timestamp"] = df["timestamp"].fillna(pd.Timestamp("2025-09-25 00:00"))

    # 3) Handle missing numeric values: fill with column mean
    for col in ["call_minutes", "data_gb"]:
        if df[col].isna().sum() > 0:
            mean_val = df[col].mean()
            df[col] = df[col].fillna(mean_val)
            logging.info("Filled missing values in %s with mean=%.2f", col, mean_val)

    # 4) Aggregate total per customer
    agg = (
        df.groupby(["customer_id", "region"])
          .agg(
              total_call_minutes=("call_minutes", "sum"),
              total_data_gb=("data_gb", "sum"),
              last_activity=("timestamp", "max"),
          )
          .reset_index()
    )

    # 5) Flag heavy users
    agg["heavy_user"] = (agg["total_data_gb"] > 50) | (agg["total_call_minutes"] > 500)

    logging.info("Transformed data to %d aggregated customer rows", len(agg))
    return agg

# Use the function, save cleaned data to a NEW CSV, and print it ---

df_raw = extract_raw_data("telecom_raw.csv")   # raw CSV is only read, not modified
df_clean = transform_and_clean(df_raw)

# Save cleaned/aggregated data to a separate CSV
clean_csv_path = "telecom_cleaned.csv"
df_clean.to_csv(clean_csv_path, index=False)

print("Cleaned dataset (first few rows):")
print(df_clean.head())

print(f"\nCleaned data has been saved to: {clean_csv_path}")
```

You should see one row per customer with total_call_minutes, total_data_gb, last_activity, and heavy_user.

5. Load to SQLite “warehouse”

We now store the cleaned data in a SQLite DB and create an index on customer_id.

```
def load_to_warehouse(df_clean, db_path=DB_PATH):
    """
    Load cleaned data into SQLite as 'usage_summary' table.
    Creates an index on customer_id for fast lookups.
    """
    conn = sqlite3.connect(db_path)
    cur = conn.cursor()

    # Write DataFrame to table (replace for each run)
    df_clean.to_sql("usage_summary", conn, if_exists="replace", index=False)

    # Create index on customer_id
    cur.execute("CREATE INDEX IF NOT EXISTS idx_usage_summary_cust ON
usage_summary(customer_id);")

    conn.commit()
    conn.close()
    logging.info("Loaded %d rows into usage_summary table", len(df_clean))

load_to_warehouse(df_clean)

# Quick check: read from DB and show
conn = sqlite3.connect(DB_PATH)
check_df = pd.read_sql_query("SELECT * FROM usage_summary;", conn)
conn.close()

print("Data inside SQLite usage_summary table:")
check_df
```

You should see the same cleaned rows coming from SQLite.

6. Apply Governance (role-based views)

We’ll create two SQL views:
vw_usage_junior – hides names/phones
vw_usage_manager – full usage table.

For this mini-project we’ll simulate PII fields (customer_name, phone) inside SQLite and show how views would mask them.

```
#Enrich table with fake PII fields

def enrich_with_pii(db_path=DB_PATH):
    """
    Adds fake PII columns to usage_summary to demonstrate role-based masking.
    """
    conn = sqlite3.connect(db_path)
    cur = conn.cursor()
```

```python
    # Add columns if they don't exist
    cur.execute("PRAGMA table_info(usage_summary);")
    cols = [row[1] for row in cur.fetchall()]

    if "customer_name" not in cols:
        cur.execute("ALTER TABLE usage_summary ADD COLUMN customer_name TEXT;")
    if "phone" not in cols:
        cur.execute("ALTER TABLE usage_summary ADD COLUMN phone TEXT;")

    # Simple mapping for demo
    name_map = {
        1001: "Asha Mehta",
        1002: "Ravi Kumar",
        1003: "Sneha Rao",
        1004: "Manoj Singh",
        1005: "Divya Jain",
        1006: "Rahul Roy",
        1007: "Neha Gupta",
    }

    # Update rows with names and fake phone numbers
    for cust_id, name in name_map.items():
        cur.execute(
            """
            UPDATE usage_summary
            SET customer_name = ?, phone = ?
            WHERE customer_id = ?;
            """,
            (name, f"9{cust_id}000000", cust_id)
        )

    conn.commit()
    conn.close()
    logging.info("Enriched usage_summary with fake PII fields.")

enrich_with_pii()
```

Next Step – Create role-based SQL views

```python
def create_role_based_views(db_path=DB_PATH):
    """
    Create role-based views:
    - vw_usage_junior: masked PII
    - vw_usage_manager: full access
    """
    conn = sqlite3.connect(db_path)
    cur = conn.cursor()

    # Junior view: NO real name/phone, only masked info
    cur.execute("""
        CREATE VIEW IF NOT EXISTS vw_usage_junior AS
```

```
        SELECT
            customer_id,
            region,
            total_call_minutes,
            total_data_gb,
            last_activity,
            heavy_user,
            'ANON' AS customer_name,
            'XXXXXX' AS phone
        FROM usage_summary;
    """)

    # Manager view: full details
    cur.execute("""
        CREATE VIEW IF NOT EXISTS vw_usage_manager AS
        SELECT
            customer_id,
            region,
            total_call_minutes,
            total_data_gb,
            last_activity,
            heavy_user,
            customer_name,
            phone
        FROM usage_summary;
    """)

    conn.commit()
    conn.close()
    logging.info("Created role-based views for junior and manager roles.")

create_role_based_views()
```

Next Step – Helper function to read data for a given role

```
def get_usage_for_role(role, db_path=DB_PATH, limit=10):
    """
    Utility: return a DataFrame for the requested role ('junior' or 'manager').
    """
    view_map = {
        "junior": "vw_usage_junior",
        "manager": "vw_usage_manager",
    }
    view_name = view_map.get(role.lower())
    if view_name is None:
        raise ValueError("Role must be 'junior' or 'manager'")

    conn = sqlite3.connect(db_path)
    df = pd.read_sql_query(f"SELECT * FROM {view_name} LIMIT {limit};", conn)
    conn.close()
    return df
```

```
print("Junior analyst view:")
display(get_usage_for_role("junior"))

print("\nManager view:")
display(get_usage_for_role("manager"))
```

You should see that the junior view has customer_name = 'ANON' and phone = 'XXXXXX', while the manager view shows real names and phone numbers.

7. Automate the pipeline (every 15 seconds)

Let's wrap everything into a single function run_pipeline() and then schedule it. Define the full pipeline function:

```
def run_pipeline():
    """
    Runs the full pipeline once:
    1) Extract raw data from CSV
    2) Transform & clean
    3) Load to warehouse (SQLite)
    4) Enrich with PII and create role-based views
    """
    try:
        logging.info("Starting ETL pipeline run...")

        # 1 & 2: Extract + Transform
        df_raw = extract_raw_data("telecom_raw.csv")
        df_clean = transform_and_clean(df_raw)

        # 3: Load
        load_to_warehouse(df_clean, DB_PATH)

        # 4: Governance
        enrich_with_pii(DB_PATH)
        create_role_based_views(DB_PATH)

        logging.info("ETL pipeline run completed successfully.")

    except Exception as e:
        logging.exception("ETL pipeline run FAILED: %s", e)
        print("Pipeline failed, see log file for details.")
```

Next Step – Test running the pipeline once

```
run_pipeline()

print("Manager view after pipeline run:")
get_usage_for_role("manager")
```

If this works and the table/views look correct, you're ready to automate.

8. Scheduling the pipeline (simulated every 20 seconds)

For classroom demos, let it run for 3 iterations then stop it. Schedule job every 20 seconds

```
import time, schedule

schedule.clear()
schedule.every(20).seconds.do(run_pipeline)

runs = 3  # run 3 times then stop
for _ in range(runs):
    schedule.run_pending()
    time.sleep(10)   # wait for the next tick

print("✅ Done. Scheduler exited after", runs, "runs.")
```

Watch the notebook; every ~15 seconds, a new log entry will be added in telecom_pipeline.log and the SQLite file telecom_warehouse.db will be updated.

9. Connecting SQLite with Power BI for Visualization
Follow the steps below to connect your telecom_warehouse.db SQLite database to Power BI and build visual dashboards using the cleaned dataset and governance views.

Step 1 — Install SQLite ODBC Driver
Power BI does not natively support SQLite, so an ODBC connector is required.

- Download a **64-bit SQLite ODBC driver** (matching Power BI Desktop architecture).
- Install using default settings.
- Restart Power BI if already open.

Step 2 — Open Power BI Desktop

- Launch **Power BI Desktop**
- Select **Get Data → More...**

Step 3 — Choose ODBC Source

- Search for **ODBC**
- Click **Connect**

Step 4 — Select or Create SQLite Data Source
If a SQLite DSN already exists, select it.
If not:

1. Click **DSN Manager**
2. Create a new DSN
3. Select: **SQLite ODBC Driver**
4. Browse and point it to:

telecom_warehouse.db
Click **OK**.

Step 5 — Select the Table or View
Power BI will list all database objects including:

Type	Name	Access Level
Table	usage_summary	Full cleaned dataset
View	vw_usage_junior	Masked data (no PII)
View	vw_usage_manager	Full PII access

Choose based on reporting/audience needs.
Then click:
➞ Load

Step 6 — Build Visuals
Once data loads, you can create visualizations such as:

- **Bar Chart:** Top heavy users
- **Stacked Chart:** Data usage by region
- **KPI Cards:** Total call minutes, total data consumption
- **Filters:** Region, heavy user flag, time period

10. What students should submit

1. telecom_raw.csv
2. telecom_warehouse.db
3. Notebook telecom_pipeline.ipynb
4. Screenshot of:
   - Junior vs manager views, or
   - Power BI dashboard