Name: Aditya Sharma
Roll No.: CH21B006

DA5402: Assignment #2

Module 1 [5 points]
Create a suitable Airflow Operator that would either import your Python function or directly execute your Python script to scrape the home page of Google News. The URLs to scrape may change over time, so you should keep them as configurable parameters or operator properties.

- Function Signature: run_module1(config_path: str) is designed so Airflow's PythonOperator can call it directly, passing in the path to your configuration file.
- Configuration Loading: It first calls load_config(config_path), which reads pipeline_config.ini (or defaults) and returns a dictionary of settings—making the code flexible and easy to update without changing Python logic.
- Selenium Scraping: Inside run_module1, it calls access_homepage(config), initialising a headless Chrome browser and navigating to Google News. This step ensures the homepage is accessible and captures a minimal verification (page title, timestamp).
- Output to JSON: After scraping, the function writes a simple JSON file (module1_metadata.json) containing the homepage URL, page title, and timestamp. This metadata can be used by subsequent modules in the pipeline.

- Also, selenium needs to be added to the Docker airflow image, so that it can be integrated into airflow.
- Installing Selenium via requirements.txt: We list selenium==4.9.1 in our requirements.txt, ensuring the Python dependency is installed during the image build.
- Chrome/Chromium in the Dockerfile: The Dockerfile installs Chromium (or Google Chrome) plus the matching driver (chromium-driver or chromedriver) so Selenium can run in headless mode.
- Headless Environment: In the Dockerfile, we also add flags like --no-sandbox and --disable-dev-shm-usage, allowing Chrome/Chromium to run reliably without a full GUI in the container.

Module 2 [5 points]
Create a suitable Airflow Operator that would either import your Python function or directly executing your Python script to scrape the "Top stories" from the home page of Google News. The URLs to scrape may change over time, so you have to keep them as configurable parameters or operator properties.

- run_module2 locates the “Top stories” link on Google News using an XPath selector and Selenium.
- It writes that extracted URL to top_stories_link.txt and also stores metadata (timestamp, URL, and XPath used) in a JSON file.
- This ensures subsequent modules know exactly where to find the Top Stories page.

Module 3 [10 points]
Create a suitable Airflow Operator that would either import your Python function or directly executing your Python script to extract the thumbnail and the headline of every story from that page. Remember that the page is set up for lazy loading. Your operator should factor lazy loading. The layout of the “Top stories” page may change over time, so ensure that you create your Operator in a way for easy updates later.

- run_module3 first retrieves the Top Stories URL (either from Module 2’s output file or a default), then opens it in a headless Selenium browser.
- It scrolls multiple times to ensure lazy-loaded articles are visible, then uses BeautifulSoup to parse the page’s HTML.
- Each headline and thumbnail is extracted and stored as a JSON list (module3_stories.json), along with module-level metadata in a separate file.
- This data is then available to later modules, providing a complete set of story details (headline, article link, image URL) for further processing.

Module 4 [10 points]
Create a Postgres operator to setup your database tables with the necessary unique/primary keys. Your database should have one table for storing the image data (base64 encoded) and the other table for storing the headlines and other meta information such as URLs, scrape timestamp, article date, etc. You can run a join query with the image and headlines tables to fetch the pair. After the “insert” operation, you should create a status file at “dags/run/status”, which should contain just one number representing the number of successful inserts. If all the records are duplicates, you will have a ‘0’ in the status file.

- **Configuration & Database Connection:** In run_module4, we load database credentials (host, port, user, etc.) from pipeline_config.ini and connect to Postgres using psycopg2 and the tutorials_pg_conn connection in airflow. This ensures the module is flexible and can be easily updated without changing Python code.

- **Reading Stories:** We read the list of stories (extracted by Module 3) from data/module3_stories.json. This JSON contains each story's headline, link, thumbnail URL, and timestamp.
- **Insert Logic & Deduplication:** The function insert_story inserts each story into the meta_data table with an ON CONFLICT ON CONSTRAINT unique_story DO NOTHING clause. This prevents duplicates if the headline is already in the database.
- **Base64 Image Storage:** If the story is new (i.e., not a duplicate), we download the thumbnail image and base64-encode it, inserting it into the image_data table via a foreign key (meta_id) that references the new row in meta_data.
- **Status File:** After processing all stories, run_module4 writes the total number of newly inserted rows to dags/run/status. Other modules (or a second DAG) can then read this file to see if any new data arrived.
- **PostgresOperator for Table Creation:** In the Airflow DAG, we typically use a PostgresOperator to run SQL that creates meta_data and image_data tables (with unique/primary keys). This keeps table setup logic in the DAG rather than in Python code.

Adding Postgres Support in Docker:

- psycopg2-binary is listed in requirements.txt so we can connect to Postgres from Python.
- We add apache-airflow-providers-postgres in the Dockerfile so Airflow can use the PostgresOperator and other Postgres features without dependency issues.

Module 5 [10 points]
Setup a Airflow DAG to orchestrate the workflow pipeline with necessary properties and load them up in the Airflow console. You should setup crontab, to make it runnable every hour. As long as the Airflow container is running in the background, you will keep scrapping the articles for <image, headline> tuple.

The dag file named a2_dag1 has been created with the following properties:

- **Hourly Scheduling:** The schedule_interval='@hourly' ensures this entire pipeline runs automatically every hour, scraping new articles on a regular cadence.
- **Module 1 → 2 → 3:** Each is a PythonOperator that calls run_moduleX, passing the same config file. They run in a linear chain, providing data for subsequent steps.
- **Table Creation via PostgresOperator:** After Module 3, the DAG executes create_tables using SQL statements that build the meta_data and image_data tables with a unique constraint on headline and a foreign key (meta_id).
- **Unique Headline Constraint:** The UNIQUE (headline) constraint in meta_data prevents duplicates if the same headline appears multiple times.

- **Image Table Linking:** image_data references meta_data.id via meta_id, enabling a join query to fetch (headline, base64_image) pairs.
- **Module 4:** Inserts newly scraped stories into the database, storing images as base64 in image_data, then writes a status file with the number of newly inserted rows.
- **Task Dependencies:** The DAG enforces a strict order: Modules 1–3 must finish before creating tables, which must finish before Module 4 runs.
- **Reusability & Config:** All tasks read from pipeline_config.ini (in DEFAULT_CONFIG_PATH), making it easy to adjust parameters without editing Python code.

Below, is how the graph structure looks after a successful run of the dag.

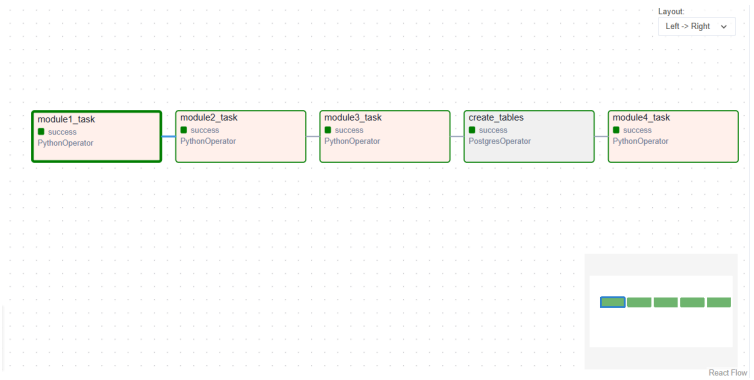

Fig. 1: Graph structure for DAG1 - a2_dag1

The logs for each module are as follows:

- **Module 1:**

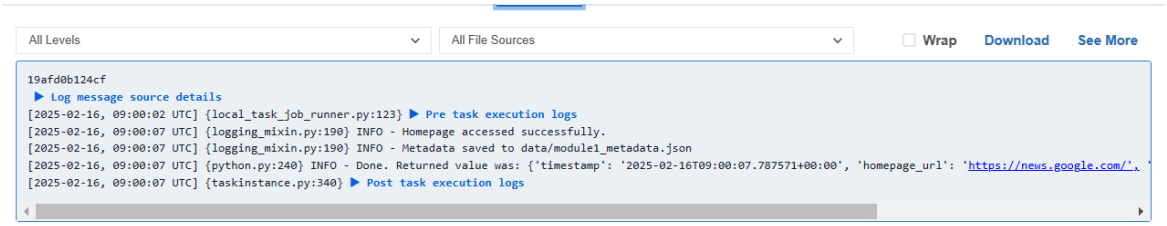

Fig. 2: module1_task airflow logs

- **Module 2:**

Fig. 3: module2_task airflow logs

- **Module 3:**

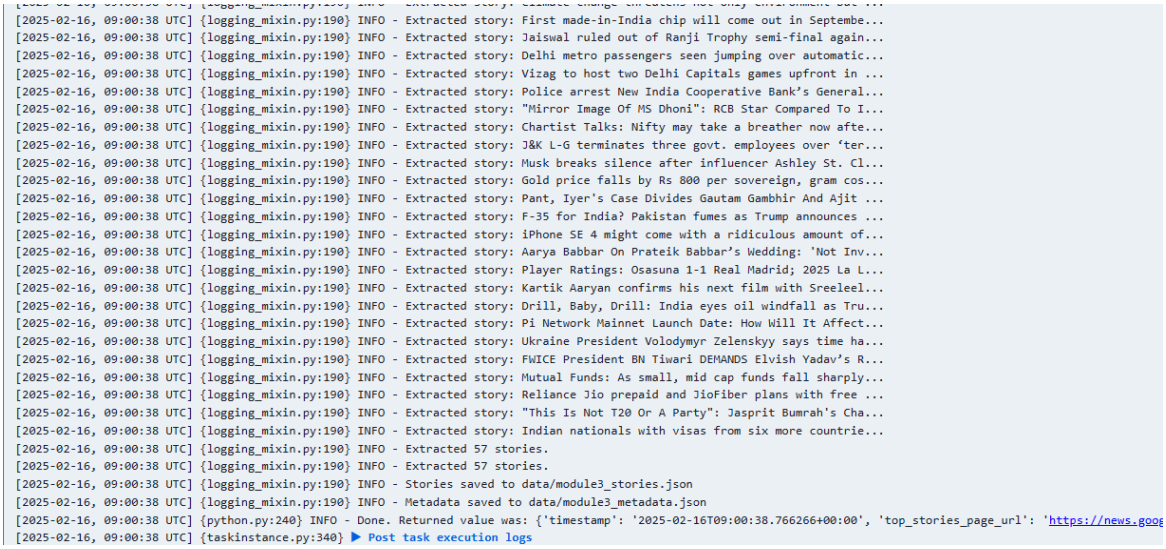

Fig. 4: module3_task airflow logs

Note that this isn't the entire log, there are logs for the rest of the stories that aren't in this screenshot.

- **Module 4:**

- Firstly there was a run that extracted all the stories from the top stories section initially as there were none stored in the database in the beginning.
- Just after completing the above run, I triggered the dag again manually, and this was the log:

Fig. 5: module4_task airflow logs(1)

- 0 new stories were inserted into the database as the dag was run straight after the previous run.
- The status file matched this by giving 0. The below image shows my status file in vs code along with the file structure that I have used.

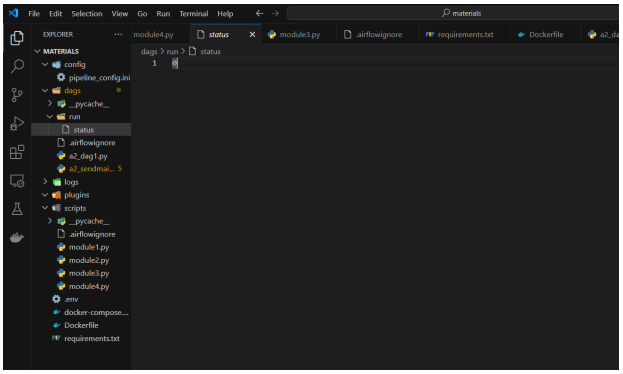

Fig. 6: status file showing 0

- If the dag was triggered after sometime, the log was a s follows:

```
19afd0b124cf
 ▶ Log message source details
[2025-02-16, 09:00:49 UTC] {local_task_job_runner.py:123} ▶ Pre task execution logs
[2025-02-16, 09:00:58 UTC] {logging_mixin.py:190} INFO - Inserted 17 new stories out of 57 total.
[2025-02-16, 09:00:58 UTC] {python.py:240} INFO - Done. Returned value was: 17
[2025-02-16, 09:00:58 UTC] {taskinstance.py:340} ▶ Post task execution logs
```

Fig. 7: module4_task airflow logs(2)

- 17 new stories were added due to the addition of new stories in the top stories page n Google News.

Module 6 [10 points]
Setup another Airflow DAG that would send an email to yourself, whenever a new <image, headline> tuple is added to the database. This DAG has to remember the previous state of the database tables to detect new rows. When news rows are detected, it should it's knowledge of the previous state and also send an email to your inbox. To send emails, you may have to learn about SMTP + TLS/SSL setup with credentials. The set up is the same process when you configure your email client (Thunderbird, Outlook, etc) to send emails from your gmail or institute accounts. The only difference is you are going to do it programmatically.
Hint: The sendmail DAG could even be one Airflow Operator. But, I want you to get creative with the modularization of the send email workflow.
This send email DAG should get triggered by the first DAG through "FileSensor" on the status file. This means, when the first DAG completes, the sendmail DAG should get triggered. The Sendmail DAG will delete the status file upon completion.

A dag named a2_sendmail_dag2 was created to do this task. It includes:

- FileSensor (wait_for_status) continuously checks for the dags/run/status file, which is created by the first DAG (Modules 1–4). Once it appears, the sensor succeeds and moves on.
- read_status (a PythonOperator) opens that file and reads the integer value indicating how many new rows were inserted in the database.
- BranchPythonOperator (branch) then decides if the DAG should send an email (send_email) or skip it (no_email) by comparing the integer to zero.
- EmailOperator (send_email) sends a notification email only if new rows were inserted (i.e., value > 0). This requires basic SMTP configuration in Airflow if you want actual emails.
- EmptyOperator (no_email) is a do-nothing placeholder if there are no new rows. This path simply skips the email step.
- cleanup_status (PythonOperator) deletes the status file so the next hourly run starts fresh. By using trigger_rule='none_failed', it always runs after either send_email or no_email, ensuring the file is cleaned up regardless of which branch was taken.
- The task dependencies form a linear flow: FileSensor → read_status → branch → [send_email, no_email] → cleanup_status.
- Because the first DAG writes the status file, this second DAG effectively monitors that file on an hourly schedule, sending emails only when new data is detected.

Here is a screenshot of how the graph looks after this dag is run successfully:

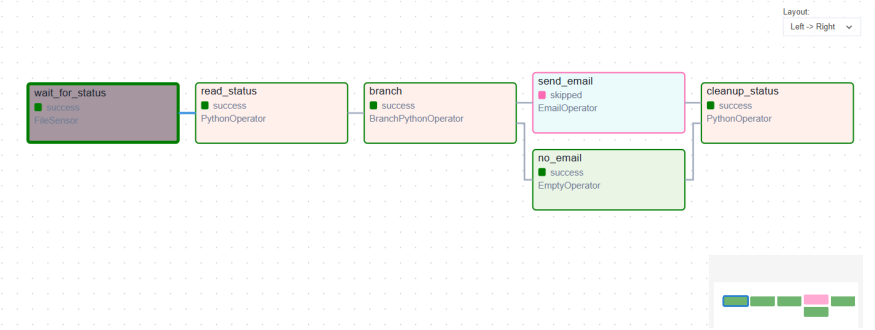

Fig. 8: Graph structure for DAG2 - a2_sendmail_dag2

- The FileSensor and PythonOperators all show green (success), indicating the DAG ran smoothly, and the BranchPythonOperator chose the “no_email” path (hence the pink “send_email” was skipped).
- A pink (skipped) box means that a particular branch wasn’t taken; in this case, no new rows were detected, so the email step was unnecessary.
- Despite one branch being skipped, the cleanup_status task still runs due to its trigger_rule='none_failed', ensuring the status file is always removed after the DAG completes.