Project Report: Dockerized Stock Data Pipeline

1. Project Overview and Executive Summary

This report details the design, implementation, and operation of a robust data pipeline for fetching, processing, and storing real-time stock market data. The system is built using a containerized approach with Docker and Apache Airflow, ensuring the pipeline is portable, scalable, and resilient to failures. The primary objective is to automate the data ingestion process from a public API into a structured database, providing a reliable foundation for data analysis and reporting. The report covers the function of each project file, a detailed line-by-line code explanation, a description of the operational flow, and a final data flow diagram.

2. Detailed Component and File Explanations

The project is structured around several interconnected files, each serving a critical role in the pipeline's lifecycle.

A. docker-compose.yml

This YAML file is the central command center for the entire project. It defines a multi-container Docker application, specifying the services, their configurations, and how they interact. It allows you to build and run the entire pipeline with a single command.

```
# This docker-compose.yml file defines the services for a new stock data pipeline.
# It sets up a PostgreSQL database and all the necessary components for an Airflow instance.


# The `services` key defines the different containers that will be created.
services:
  postgres:
    # Use the official PostgreSQL image. `postgres:14` specifies the version.
    image: postgres:14
    # A user-friendly name for the container, making it easier to identify.
    container_name: stock-postgres
    # `environment` sets environment variables inside the container for the database.
    environment:
      - POSTGRES_USER=stock_user
      - POSTGRES_PASSWORD=stock_password
      - POSTGRES_DB=stock_data
    # `ports` maps a port on your host machine to a port inside the container.
    # "5434:5432" maps host port 5434 to container port 5432, preventing conflicts.
    ports:
      - "5434:5432"
    # `volumes` is used to persist data. The named volume `stock_postgres_data`
    # ensures your database data is not lost when the container is stopped.
    volumes:
      - stock_postgres_data:/var/lib/postgresql/data
    # `healthcheck` checks if the database is ready to accept connections before other services start.
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U stock_user"]
      interval: 5s
      timeout: 5s
      retries: 5

  airflow-webserver:
    # `build: .` tells Docker to build an image from the `Dockerfile` in the current directory.
    build: .
    # A user-friendly name for the webserver container.
    container_name: stock-airflow-webserver
    # Maps the webserver's UI port to the host. Access it at http://localhost:8082.
    ports:
      - "8082:8080"
    # The `command` to run inside the container when it starts. `webserver` starts the Airflow UI.
    command: webserver
    # Passes environment variables to the container. The `AIRFLOW_VAR_` prefix
    # makes these variables available in Airflow's environment for DAGs to use.
    environment:
      - AIRFLOW__CORE__LOAD_EXAMPLES=false
      - AIRFLOW_DATABASE_SQL_ALCHEMY_CONN=postgresql+psycopg2://stock_user:stock_password@postgres/stock_data
      - AIRFLOW_VAR_STOCK_API_KEY={your free api}
      - AIRFLOW_VAR_FERNET_KEY={your fernet key }
      -
AIRFLOW__WEBSERVER__SECRET_KEY={your secret key}
    # Mounts the local `dags` folder into the container, so Airflow can find the DAG file.
    volumes:
      - ./dags:/opt/airflow/dags
      - ./data:/opt/airflow/data
    # `depends_on` ensures that the `postgres` and `airflow-init` services are
    # healthy or completed before starting the webserver.
    depends_on:
      postgres:
        condition: service_healthy
      airflow-init:
        condition: service_completed_successfully

  airflow-scheduler:
    # Similar to the webserver, it builds an image from the `Dockerfile`.
    build: .
    # A user-friendly name for the scheduler container.
    container_name: stock-airflow-scheduler
    # The `scheduler` command starts the component responsible for triggering DAG runs.
    command: scheduler
    # Passes the same environment variables needed by the DAG.
    environment:
      - AIRFLOW__CORE__LOAD_EXAMPLES=false
      -
AIRFLOW__DATABASE__SQL_ALCHEMY_CONN=postgresql+psycopg2://stock_user:st
ock_password@postgres/stock_data
```

```
      - AIRFLOW_VAR_STOCK_API_KEY={your free api}
      - AIRFLOW_VAR_FERNET_KEY={your fernet key }
    # Mounts the same local folders.
    volumes:
      - ./dags:/opt/airflow/dags
      - ./data:/opt/airflow/data
    # The scheduler also depends on the database and the initialization service.
    depends_on:
      postgres:
        condition: service_healthy
      airflow-init:
        condition: service_completed_successfully

  airflow-init:
    # Builds an image from the `Dockerfile`.
    build: .
    # Name for the initialization container.
    container_name: stock-airflow-init
    # This command runs database migrations and creates the default Airflow user.
    # It ensures the database is set up correctly for Airflow.
    command: bash -c "airflow db migrate && airflow users create --username airflow --
password airflow --firstname Airflow --lastname Airflow --role Admin --email
airflow@example.com"
    # Inherits the database environment variables.
    environment:
      - AIRFLOW__CORE__LOAD_EXAMPLES=false
      -
AIRFLOW__DATABASE__SQL_ALCHEMY_CONN=postgresql+psycopg2://stock_user:st
ock_password@postgres/stock_data
```

```
      -
AIRFLOW__WEBSERVER__SECRET_KEY={your secret key}
    # Waits for the `postgres` container to be healthy.
    depends_on:
      postgres:
        condition: service_healthy

# Defines a named volume to persist PostgreSQL data.
volumes:
  stock_postgres_data:
```

B. Dockerfile

This file provides the instructions for building the custom Docker image for our Airflow services.

```
# Dockerfile for Apache Airflow.
# This file is used by docker-compose to build the Airflow container image.

# Starts with an official Airflow base image. This provides a pre-configured
# environment with Airflow and its dependencies, saving significant time.
FROM apache/airflow:2.7.0

# Installs Python dependencies from a requirements file.
# This ensures that your Airflow environment has the necessary libraries for your DAG.

# The `COPY requirements.txt .` command copies the requirements file from your
# local directory into the root directory of the container image.
COPY requirements.txt .

# The `RUN pip install` command executes inside the container, installing all
# the Python packages listed in requirements.txt. `--no-cache-dir` is used
# to prevent caching, reducing the final image size.
RUN pip install --no-cache-dir -r requirements.txt

# The ENTRYPOINT and CMD are inherited from the base image and should not be changed.
# The base image already sets up the Airflow environment correctly.
```

C. stock_data_dag.py

This is the Python file that defines the Directed Acyclic Graph (DAG) for the pipeline. It's the "brain" of the operation, dictating the flow and scheduling of tasks.

```
# This file defines the Apache Airflow Directed Acyclic Graph (DAG) for the pipeline.
# The DAG defines the sequence of tasks to be executed.

# Import necessary libraries and classes.
import os
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.utils.dates import days_ago
from datetime import timedelta
import sys

# Adds the parent directory to the Python path. This is a common practice that
# allows Airflow to import the `stock_api_script` from a sibling directory.
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

# Imports the functions from our data script that will be executed as tasks.
from dags.stock_api_script import create_table_if_not_exists, fetch_and_store_stock_data

# --- DAG Definition ---

# Defines default arguments for all tasks in the DAG. This provides a baseline
# configuration for retries, ownership, and email notifications.
```

```
default_args = {
    'owner': 'airflow', # The owner of the DAG.
    'depends_on_past': False, # Task does not depend on the success of the previous day's run.
    'email_on_failure': False, # No email sent on task failure.
    'email_on_retry': False, # No email sent on task retry.
    'retries': 1, # Retries the task once if it fails.
    'retry_delay': timedelta(minutes=5), # Waits 5 minutes before retrying.
}

# Instantiates the DAG object.
with DAG(
    'stock_data_pipeline', # A unique ID for the DAG.
    default_args=default_args,
    description='A simple data pipeline to fetch and store stock data.',
    # Defines the schedule. `timedelta(hours=1)` means the DAG runs every hour.
    schedule_interval=timedelta(hours=1),
    # Sets the start date. `days_ago(1)` means the DAG is ready to run from yesterday.
    start_date=days_ago(1),
    tags=['stock', 'api', 'data-pipeline'], # Tags for filtering and searching in the UI.
    catchup=False, # Prevents the DAG from running for all missed intervals upon activation.
) as dag:
    # --- Define Tasks ---

    # Task 1: Creates the PostgreSQL table if it doesn't exist.
    # The `PythonOperator` calls the specified Python function.
    create_table_task = PythonOperator(
        task_id='create_stock_table', # A unique ID for this task.
        python_callable=create_table_if_not_exists, # The function to execute.
        dag=dag, # Associates the task with this DAG.
    )
    # Task 2: Fetches data from the API and stores it in the database.
    fetch_data_task = PythonOperator(
        task_id='fetch_and_store_data',
        python_callable=fetch_and_store_stock_data,
        dag=dag,
    )

    # --- Set Task Dependencies ---
    # `>>` defines the order of execution. The `fetch_data_task` will only run
    # after `create_table_task` has completed successfully.
    create_table_task >> fetch_data_task
```

D. stock_api_script.py

This Python script contains the core business logic of the pipeline. It is responsible for all data-related operations, including API interaction and database management.

```
# This script contains the core logic for the data pipeline:
# 1. Fetching data from the Alpha Vantage stock market API.
# 2. Connecting to a PostgreSQL database.
# 3. Storing the data in a table.

# Imports necessary Python libraries.
import requests # For making HTTP requests to the API.
import psycopg2 # For connecting to the PostgreSQL database.
import os # For accessing environment variables.
import logging # For logging information and errors.
import json # For handling JSON data from the API.
from datetime import datetime # For getting the current timestamp.

# Configures logging to display messages at an INFO level or higher.
logging.basicConfig(level=logging.INFO)
```

```
# --- Configuration using Environment Variables ---
# Uses `os.getenv` to securely retrieve credentials from environment variables.
# This prevents hardcoding sensitive information directly in the script.
DB_HOST = os.getenv('AIRFLOW_VAR_DB_HOST', 'postgres')
DB_NAME = 'stock_data'
DB_USER = 'stock_user'
DB_PASSWORD = 'stock_password'
STOCK_API_KEY = os.getenv('AIRFLOW_VAR_STOCK_API_KEY')
STOCK_SYMBOL = 'AAPL' # Example stock symbol to fetch.

# --- Database Operations ---

def create_table_if_not_exists():
    # `conn = None` initializes a connection object.
    conn = None
    try:
        # Establishes a connection to the PostgreSQL database using the credentials.
        conn = psycopg2.connect(host=DB_HOST, database=DB_NAME, user=DB_USER,
password=DB_PASSWORD)
        cur = conn.cursor() # Creates a cursor object to execute SQL commands.
        logging.info("Successfully connected to the PostgreSQL database.")

        # SQL command to create the table.
        create_table_sql = """
        CREATE TABLE IF NOT EXISTS stock_data (
            symbol VARCHAR(10) NOT NULL,
            price DECIMAL(10, 2) NOT NULL,
            volume INT NOT NULL,
            timestamp TIMESTAMP NOT NULL,
          PRIMARY KEY (symbol, timestamp) # Ensures each symbol/timestamp combination
is unique.
        );
        """
        cur.execute(create_table_sql) # Executes the SQL command.
        conn.commit() # Commits the transaction to save the changes.
        logging.info("Table 'stock_data' checked/created successfully.")
        cur.close() # Closes the cursor.

    except (Exception, psycopg2.DatabaseError) as error:
        logging.error(f"Error creating table: {error}") # Logs any database errors.
    finally:
        if conn is not None:
            conn.close() # Always closes the connection to free up resources.

def insert_stock_data(symbol, price, volume):
    conn = None
    try:
        conn = psycopg2.connect(host=DB_HOST, database=DB_NAME, user=DB_USER,
password=DB_PASSWORD)
        cur = conn.cursor()
        logging.info("Database connection for insertion successful.")

        # Parameterized SQL query for insertion.
        insert_sql = """
        INSERT INTO stock_data (symbol, price, volume, timestamp)
        VALUES (%s, %s, %s, %s)
        ON CONFLICT (symbol, timestamp) DO NOTHING; # Prevents duplicate entries.
        """

        # Executes the query with the data.
        cur.execute(insert_sql, (symbol, price, volume, datetime.now()))
        conn.commit()
        logging.info(f"Successfully inserted data for {symbol}.")
        cur.close()

    except (Exception, psycopg2.DatabaseError) as error:
        logging.error(f"Error inserting data into table: {error}")
    finally:
        if conn is not None:
            conn.close()

# --- API Interaction and Data Extraction ---

def fetch_and_store_stock_data():
    """
    Main function to orchestrate the data fetching and storage.
    """
    logging.info("Starting data fetching and storage process...")

    # Checks if the API key is present.
    if not STOCK_API_KEY:
        logging.error("STOCK_API_KEY is not set. Please check your environment variables.")
        return

    # Constructs the API URL with the symbol and API key.
    url                                                                                    =
f'https://www.alphavantage.co/query?function=GLOBAL_QUOTE&symbol={STOCK_SY
MBOL}&apikey={STOCK_API_KEY}'

    try:
        # Makes a GET request to the API with a 10-second timeout.
        response = requests.get(url, timeout=10)
        # `raise_for_status()` raises an HTTPError for bad responses (4xx or 5xx).
        response.raise_for_status()
        data = response.json() # Parses the JSON response into a Python dictionary.
        logging.info("Successfully fetched data from API.")

        # Data extraction from the nested JSON structure.
        stock_info = data.get('Global Quote', {})
        price = stock_info.get('05. price')
        volume = stock_info.get('06. volume')

        # Error handling for missing data in the response.
        if price is None or volume is None:
            logging.warning("Missing 'price' or 'volume' in API response. Skipping insertion.")
            return

        # Type conversion and validation.
        try:
            price = float(price)
            volume = int(volume)
        except (ValueError, TypeError) as e:
            logging.error(f"Failed to convert data types: {e}. Raw data: price={price}, volume={volume}")
            return

        # Calls the function to store the processed data.
        insert_stock_data(STOCK_SYMBOL, price, volume)
    except requests.exceptions.RequestException as e:
        logging.error(f"Failed to fetch data from API: {e}")
    except (json.JSONDecodeError, KeyError) as e:
        logging.error(f"Failed to parse API response: {e}")
```

```
# This block allows the script to be run directly for testing.
if _name___== "_main_":
    create_table_if_not_exists()
    fetch_and_store_stock_data()
```

3. How the Pipeline Runs

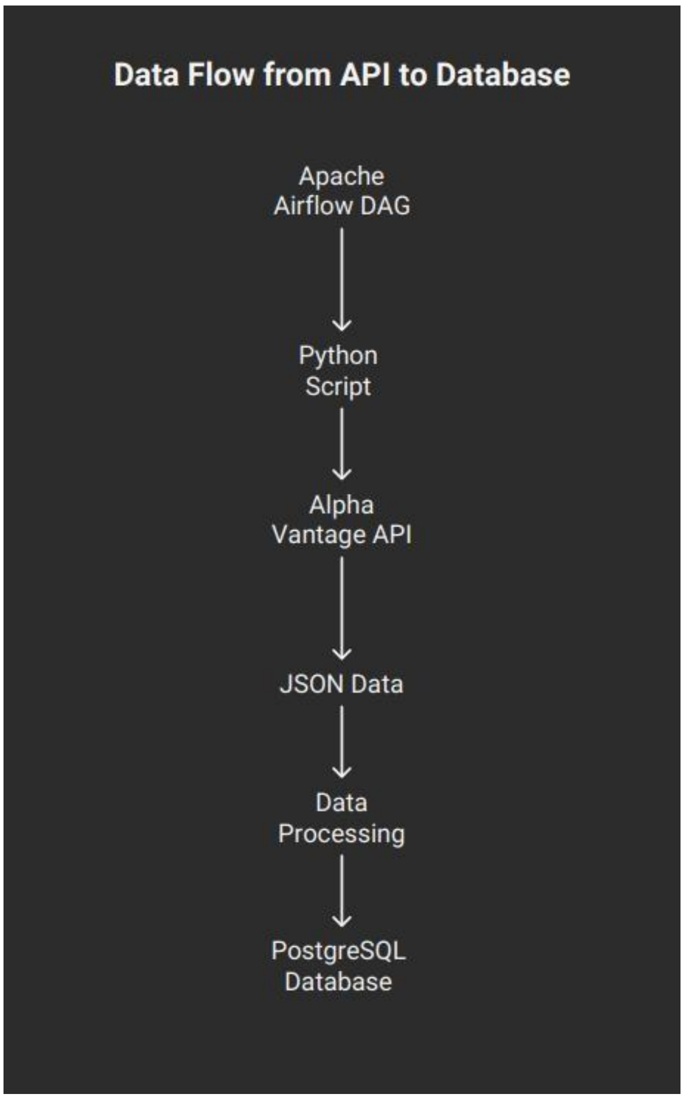
The entire pipeline is a self-contained system managed by Docker Compose. Here is a step-by-step breakdown of the operational flow from a single command:

1. **System Startup:** The docker compose up -d --build command is executed from the terminal. Docker reads the docker-compose.yml file to understand the entire system's configuration.
2. **Container Provisioning:** Docker starts the postgres container first. The healthcheck ensures that the database is fully operational and ready to accept connections.
3. **Airflow Initialization:** Once the database is healthy, the airflow-init service runs. Its purpose is to perform one-time setup tasks, such as creating Airflow's internal database tables and a default administrator user**.**
4. **Service Launch:** With the database and core setup complete, the airflow-webserver and airflow-scheduler services are started. The airflow-webserver exposes the Airflow UI, while the airflow-scheduler becomes active, scanning for DAG files.
5. **DAG Discovery**: The scheduler finds the stock_data_dag.py file within the mounted dags folder. It parses the file to understand the pipeline's structure, schedule, and dependencies.
6. **Scheduled Execution:** Based on the schedule_interval (e.g., hourly), the scheduler creates a new DAG run at the appropriate time.
7. **Task Execution:** The scheduler initiates the first task in the DAG, create_stock_table. A new container or process is spawned to run the create_table_if_not_exists() Python function.
8. **Data Ingestion:** Once create_stock_table completes successfully, the scheduler starts the next task, fetch_and_store_data. This task executes the fetch_and_store_stock_data() Python function, which performs the following actions:
    - Makes a request to the Alpha Vantage API.
    - Parses the JSON response.
    - Extracts the relevant stock data (price and volume).
    - Establishes a connection to the PostgreSQL database.
    - Inserts the processed data into the stock_data table.

9. **Data Flow Completion:** The task completes, and the data is now persistently stored in the database, ready for analysis. The entire cycle will repeat on the next scheduled interval, ensuring a continuous stream of up-to-date data.

4. Data Flow Diagram

Description: The data flow for this pipeline is a sequential process that moves information from an external source to its final destination. The flow begins with the Apache Airflow DAG, which acts as the orchestrator. When the DAG is triggered, it initiates a Python script to fetch the data. This script makes an HTTP request to the Alpha Vantage API, which is our data source. The API responds with a JSON object containing the stock data. The Python script then processes this JSON, extracting the specific data points needed, and writes this cleaned data to the PostgreSQL database, which serves as the final data sink.


Data Flow from API to Database
Apache Airflow DAG
Python Script
Alpha Vantage API
JSON Data
Data Processing
PostgreSQL Database