If you're using the **TMDb API** to build a **movie recommender system** with **Apache Airflow**, here's a structured **end-to-end approach**:

Step 1: Define the Architecture

Your workflow involves:

1. **Data Ingestion** – Extract movie data from TMDb API.
2. **Data Storage** – Store movie metadata in a distributed database (e.g., Snowflake, BigQuery, or PostgreSQL).
3. **Feature Engineering** – Transform and preprocess the data.
4. **Model Training** – Build a recommender system using collaborative filtering, content-based filtering, or hybrid models.
5. **Model Deployment & API** – Deploy a model that serves recommendations via an API.

Tech Stack

- **Apache Airflow** (Orchestrator)
- **TMDb API** (Data Source)
- **Snowflake/BigQuery/PostgreSQL** (Storage)
- **Spark/Pandas** (Processing)
- **Scikit-learn/Surprise/LightFM** (Modeling)
- **Flask/FastAPI** (API for serving recommendations)

Step 2: Set Up Apache Airflow DAG for Data Ingestion

The first DAG will:

- Query TMDb API daily for new/updated movies.
- Store the data in a database.

Tasks for Airflow DAG

1. **Extract Data from TMDb API**
   - Use `requests` to fetch movie metadata.
   - Query popular, trending, and newly released movies.

2. **Transform Data**
    - Normalize JSON response.
    - Extract relevant fields (movie ID, title, genre, cast, etc.).
3. **Load Data into a Database**
    - Store the movie data in **Snowflake, PostgreSQL, or BigQuery**.

Airflow DAG Code (ETL Pipeline)

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta
import requests
import pandas as pd
import sqlalchemy

# TMDb API Config
API_KEY = "your_tmdb_api_key"
BASE_URL = "https://api.themoviedb.org/3"

# Database Connection (PostgreSQL Example)
DATABASE_URI =
"postgresql+psycopg2://user:password@host:5432/moviedb"
engine = sqlalchemy.create_engine(DATABASE_URI)

def fetch_movies():
    """Fetch movie data from TMDb API."""
    url = f"{BASE_URL}/movie/popular?api_key=
{API_KEY}&language=en-US&page=1"
    response = requests.get(url)
    data = response.json()["results"]

    # Convert to DataFrame
    df = pd.DataFrame(data, columns=["id", "title",
"release_date", "vote_average", "genre_ids", "overview"])
    df.to_sql("movies", con=engine, if_exists="append",
index=False)

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2024, 2, 1),
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "tmdb_movie_etl",
    default_args=default_args,
```

```
    schedule_interval="@daily",
    catchup=False,
)

fetch_movies_task = PythonOperator(
    task_id="fetch_movies",
    python_callable=fetch_movies,
    dag=dag,
)

fetch_movies_task
```

Step 3: Feature Engineering

- Process the stored movie data.
- Convert genres and keywords into **one-hot encoded vectors**.
- Extract text-based features from movie descriptions (`TF-IDF`).
- Normalize ratings for similarity computation.

Example: **Processing Genres for Content-Based Filtering**

```
import pandas as pd
from sklearn.preprocessing import MultiLabelBinarizer

df = pd.read_sql("SELECT id, title, genre_ids FROM movies",
con=engine)

# Convert genre_ids to list
df["genre_ids"] = df["genre_ids"].apply(lambda x: eval(x) if
isinstance(x, str) else x)

# One-hot encode genres
mlb = MultiLabelBinarizer()
genres_encoded =
pd.DataFrame(mlb.fit_transform(df["genre_ids"]),
columns=mlb.classes_)
df = df.drop(columns=["genre_ids"]).join(genres_encoded)

df.to_sql("movies_processed", con=engine, if_exists="replace",
index=False)
```

Step 4: Build the Recommender System

You can use **two main approaches**:

1. Content-Based Filtering (Using Cosine Similarity)

- Compute similarity based on **genres, keywords, descriptions**.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Load preprocessed data
df = pd.read_sql("SELECT * FROM movies_processed", con=engine)

# Convert overview text into TF-IDF features
tfidf = TfidfVectorizer(stop_words="english")
tfidf_matrix = tfidf.fit_transform(df["overview"].fillna(""))

# Compute similarity
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)

# Store similarity matrix
import numpy as np
np.save("movie_similarity.npy", cosine_sim)
```

2. Collaborative Filtering (Using Surprise)

- Uses user ratings to recommend movies.

```
from surprise import Dataset, Reader, SVD
from surprise.model_selection import train_test_split

# Load user ratings
df_ratings = pd.read_sql("SELECT user_id, movie_id, rating
FROM ratings", con=engine)

# Prepare dataset
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(df_ratings[["user_id", "movie_id",
"rating"]], reader)

# Train model
trainset, testset = train_test_split(data, test_size=0.2)
model = SVD()
model.fit(trainset)

# Save model
import pickle
with open("collab_filtering_model.pkl", "wb") as f:
    pickle.dump(model, f)
```

Step 5: Deploy the Recommender System

Once trained, deploy a **Flask or FastAPI** service that serves recommendations.

Flask API for Movie Recommendations

```
from flask import Flask, request, jsonify
import numpy as np

app = Flask(__name__)

# Load similarity matrix
cosine_sim = np.load("movie_similarity.npy")

@app.route("/recommend", methods=["GET"])
def recommend():
    movie_id = int(request.args.get("movie_id"))
    idx = df[df["id"] == movie_id].index[0]

    # Get top similar movies
    similar_indices = cosine_sim[idx].argsort()[-6:-1][::-1]
    recommendations = df.iloc[similar_indices][["id",
"title"]].to_dict(orient="records")

    return jsonify(recommendations)

if __name__ == "__main__":
    app.run(debug=True)
```

Step 6: Automate Everything with Airflow

- **Ingestion DAG** – Fetch new data daily.
- **Processing DAG** – Feature engineering weekly.
- **Model Retraining DAG** – Train recommender model monthly.
- **API Deployment DAG** – Deploy new model when retrained.

Final Thoughts

🎯 **By following this pipeline, you'll have:**

- **Daily-updated** movie data from TMDb.
- **A structured and scalable recommender system.**
- **An Airflow-based automated pipeline.**

- **A deployed API serving recommendations.**

Would you like help setting up Airflow DAGs for **model training and deployment** as well? 🚀