**Voyage Analytics: Integrating MLOps in Travel**

**Productionization of ML Systems**

**Step 1: Exploratory Data Analysis (EDA)  
Objective**

The goal of the Exploratory Data Analysis (EDA) step is to understand the structure, quality, and patterns in the data before performing any modeling or advanced analysis. In this project, EDA helps us analyze user, flight, and hotel datasets to identify trends, relationships, missing values, and potential data issues.

**1. Importing Required Libraries**

In this step, we import essential Python libraries required for data analysis and visualization. Pandas is used for data manipulation, Seaborn and Matplotlib are used for creating visualizations, and ydata\_profiling is used to generate an automated and detailed data profiling report.

Setting the Seaborn style ensures that all plots have a clean and consistent appearance.

**2. Loading the Datasets**

Three datasets are loaded into the environment:

* Users dataset
* Flights dataset
* Hotels dataset

Each dataset is read from a CSV file using Pandas. After loading, the shape of each dataset is printed to verify the number of rows and columns, ensuring the data has been imported correctly.

**3. Initial Data Inspection**

The first few rows of each dataset are displayed using the head() function. This helps in understanding:

* Column names
* Data types
* Sample values
* Overall structure of the datasets

This quick glimpse is useful for identifying obvious data quality issues at an early stage.

**4. Automated Data Profiling**

An automated profiling report is generated for the flights dataset using ydata\_profiling. This report provides:

* Summary statistics
* Distribution of numerical features
* Correlation analysis
* Missing value analysis
* Detection of potential outliers

If the interactive widget cannot be rendered in the environment, the report is exported as an HTML file that can be viewed in a web browser.

**5. Basic Data Visualizations**

Basic visualizations are created to understand key patterns in the flights dataset:

* A histogram of flight prices to analyze price distribution
* A scatter plot of distance versus price to understand their relationship

These plots help in identifying trends, skewness, and potential anomalies in the data.

**6. Missing Values Analysis**

A heatmap is used to visualize missing values across the flights dataset. This allows easy identification of columns with missing data and helps decide whether data cleaning or imputation is required in later steps.

**Step 2: Regression Model Development & API Serving**

**Objective**

The objective is to build a robust regression model that predicts flight prices using historical travel data and to expose this model via a REST API. This step focuses on feature engineering, model training, experiment tracking, and real-time prediction readiness using production-grade practices.

**1. Environment Setup**

A Python virtual environment is activated to ensure dependency isolation and reproducibility. All required libraries are installed using a requirements.txt file, which includes Flask for API development, Pandas for data handling, Scikit-learn for machine learning, MLflow for experiment tracking, and Joblib for model serialization.

This setup ensures that the project can be consistently executed across different systems without dependency conflicts.

**2. Model Training Pipeline**

The regression model is trained using a modular training script. The training process begins by merging the users, flights, and hotels datasets into a unified dataset. This consolidated view enables richer feature representation and improved prediction accuracy.

Feature preparation is handled through a dedicated preprocessing pipeline that separates numerical and categorical variables. Categorical features are encoded, numerical features are scaled as required, and the target variable is defined as flight price.

A Random Forest Regressor is selected for its ability to handle non-linear relationships and mixed feature types. The model is trained using a Scikit-learn pipeline that combines preprocessing and model training into a single, reproducible workflow.

**3. Experiment Tracking with MLflow**

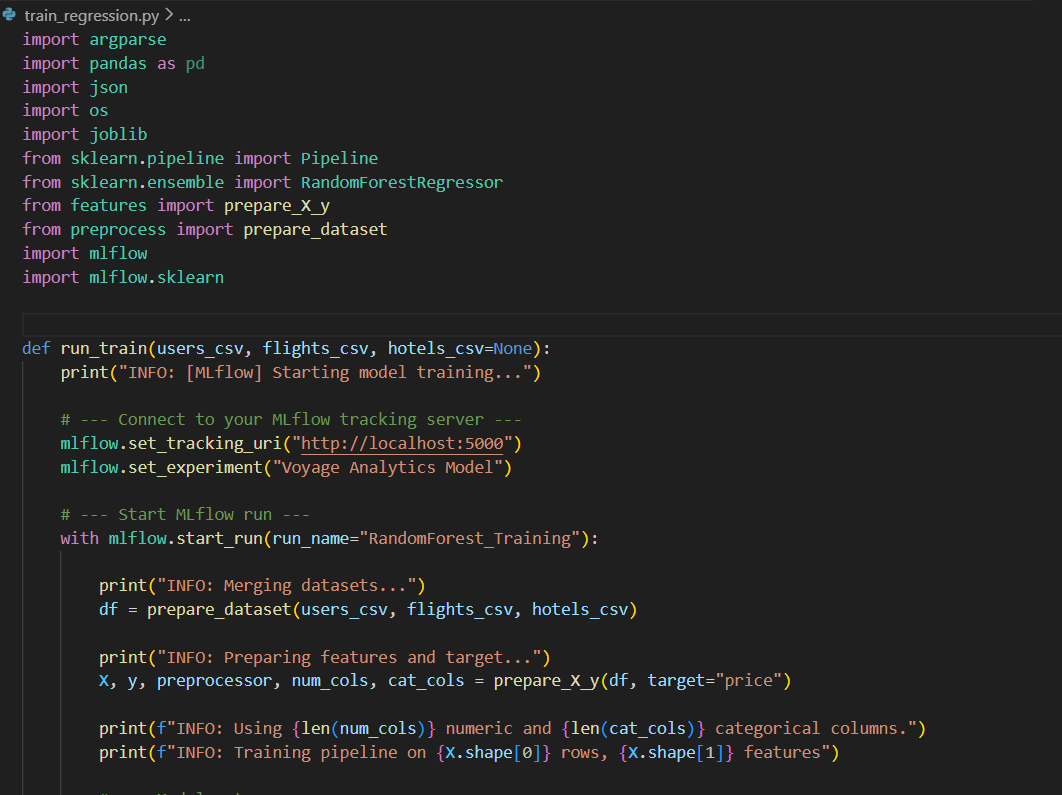
MLflow is integrated into the training pipeline to track model parameters, metrics, and artifacts. During training, hyperparameters such as the number of trees and maximum depth are logged, along with performance metrics like the R² score.

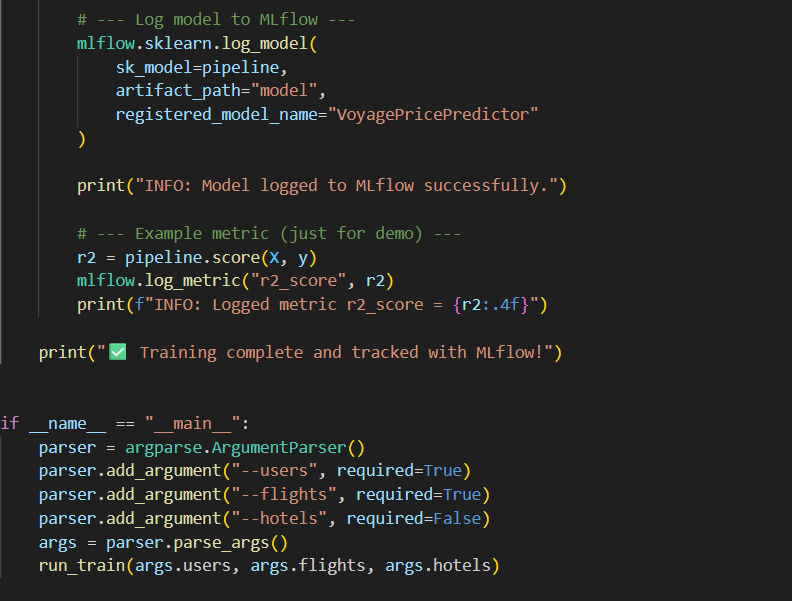
The trained model is registered in the MLflow Model Registry, enabling version control, comparison across experiments, and smooth promotion to production stages.

**4. Model Serialization and Metadata Storage**

After training, the complete pipeline is saved locally using Joblib. Along with the model, a columns.json file is generated to store metadata about the features used during training. This ensures that incoming API requests align exactly with the model’s expected input structure.

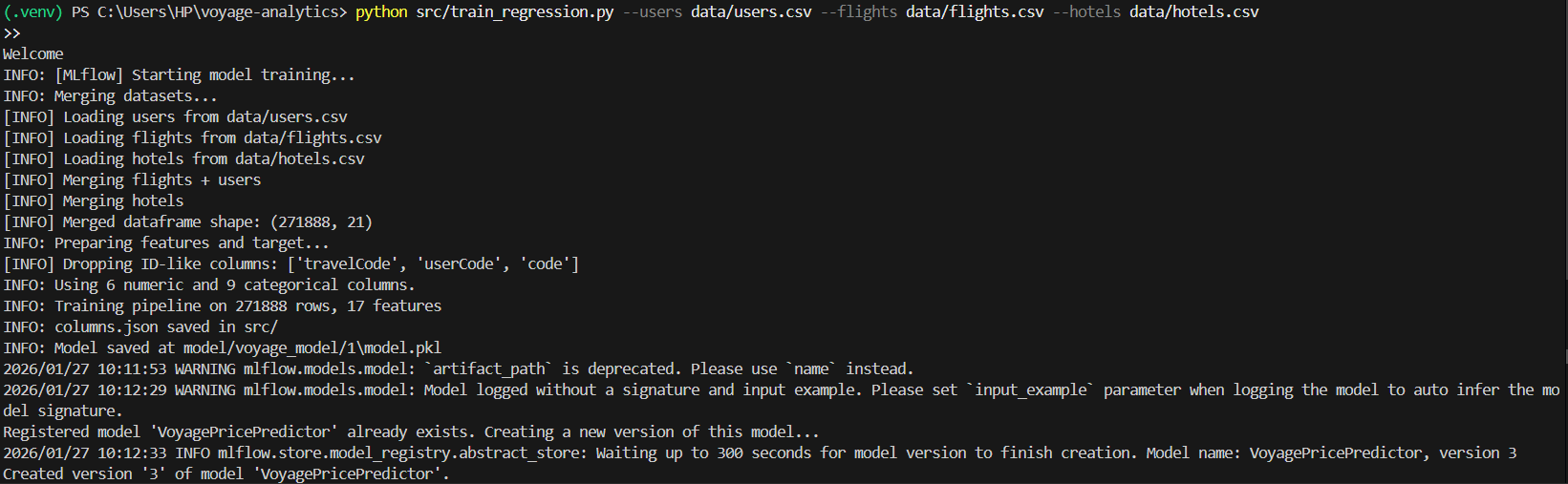
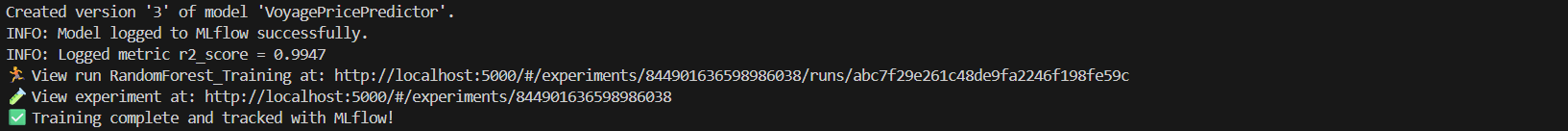
This design prevents schema mismatches during inference and improves model reliability in production.

**Code - train\_regression.py**



**Run:**

**python src/train\_regression.py --users data/users.csv --flights data/flights.csv --hotels data/hotels.csv**

****

**Code Explanation:**

**🔹 Training Script (train\_regression.py)**

* **Argument Parsing**  
  Takes file paths for users, flights, and hotels datasets from the command line, making the script reusable and automation-friendly.
* **MLflow Setup**  
  Connects to an MLflow tracking server, sets an experiment, and starts a run to track parameters, metrics, and models.
* **Dataset Preparation**  
  Merges users, flights, and hotels data into a single dataset and prepares features (X) and target (y) using preprocessing utilities.
* **Feature Engineering**  
  Separates numerical and categorical columns and applies appropriate preprocessing through a Scikit-learn preprocessor.
* **Model Pipeline**  
  Uses a Scikit-learn Pipeline combining preprocessing and a RandomForestRegressor, ensuring consistent transformations during training and inference.
* **Model Training & Logging**  
  Trains the model, logs hyperparameters and R² score to MLflow, and registers the model in the MLflow Model Registry.
* **Model Persistence**  
  Saves the trained pipeline using Joblib and stores feature metadata in columns.json to maintain schema consistency during predictions.

**REST API Development Using Flask**

A Flask-based REST API is implemented to serve the trained regression model. The application loads the model asynchronously at startup to avoid blocking server initialization.

The API exposes multiple endpoints:

* A health endpoint to verify service and model availability
* A root endpoint to confirm API status
* A prediction endpoint that accepts JSON input and returns predicted flight prices

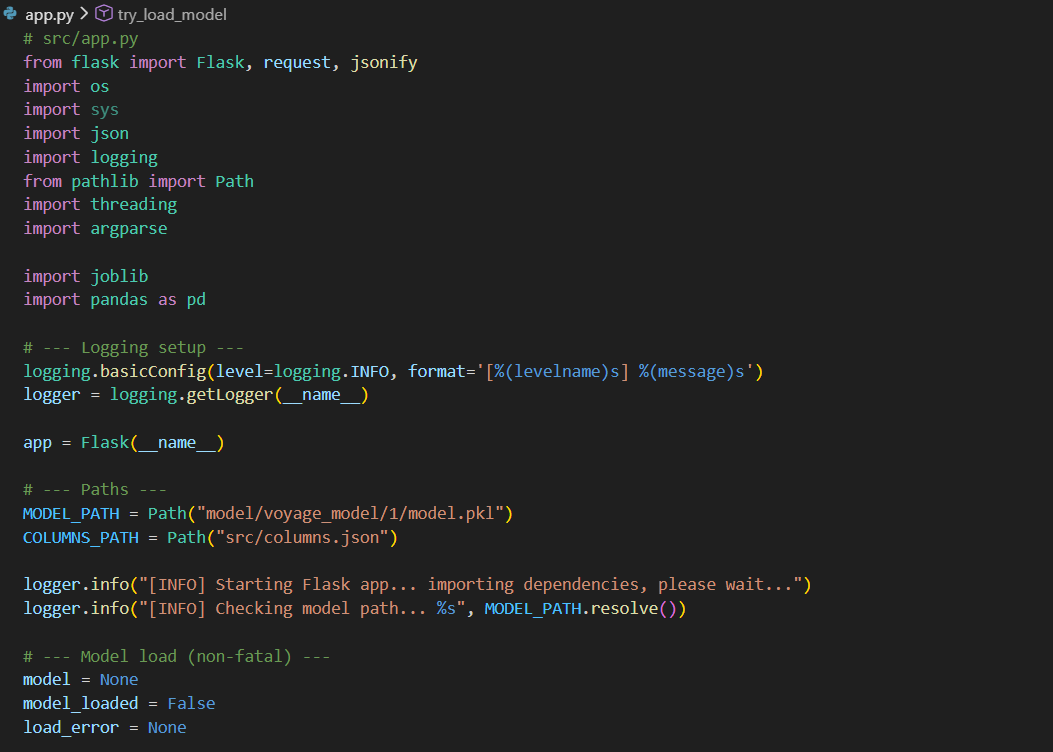
Incoming request data is validated, transformed into a structured DataFrame, and passed through the trained pipeline to generate predictions.

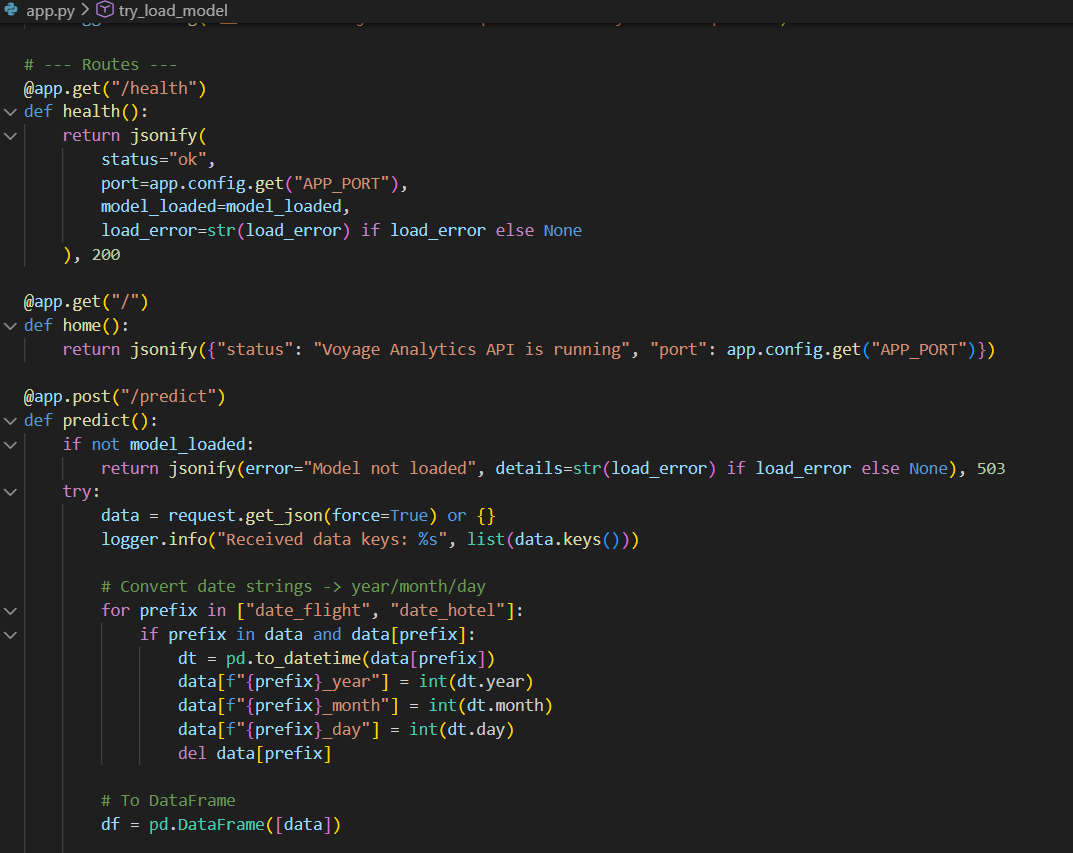
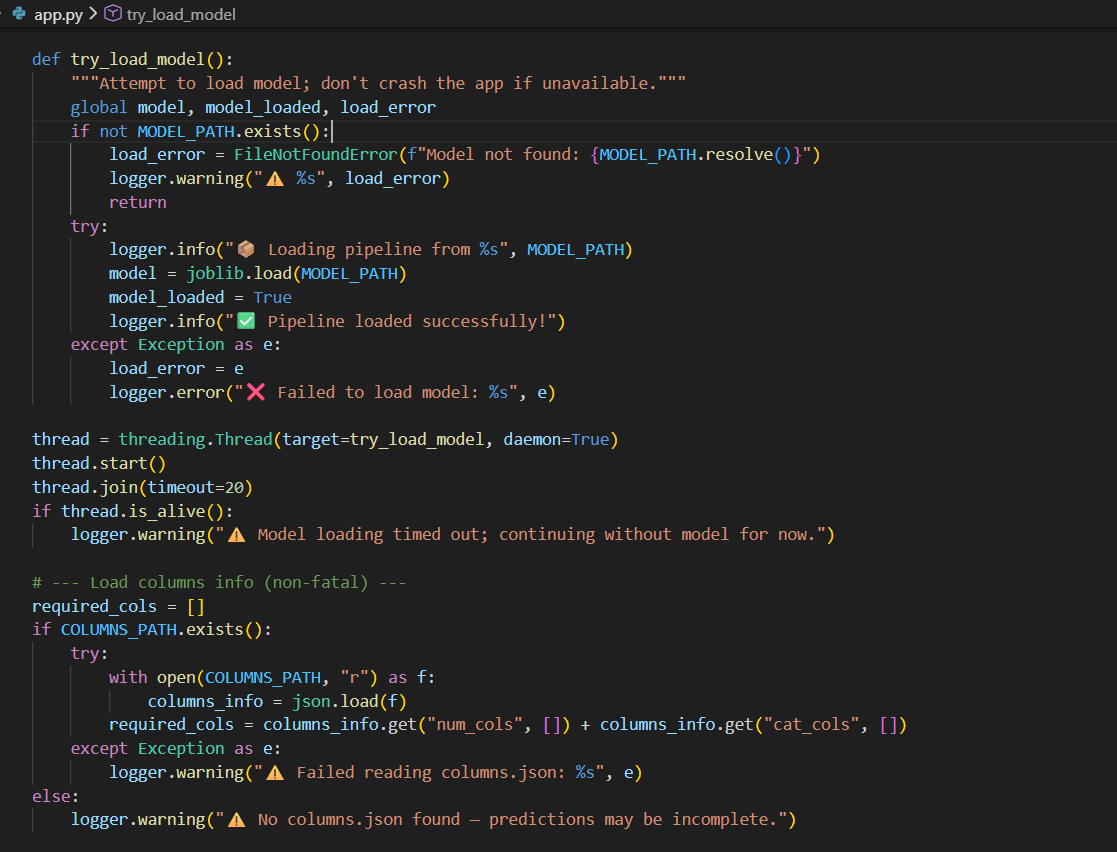
**6. Input Processing and Prediction Logic**

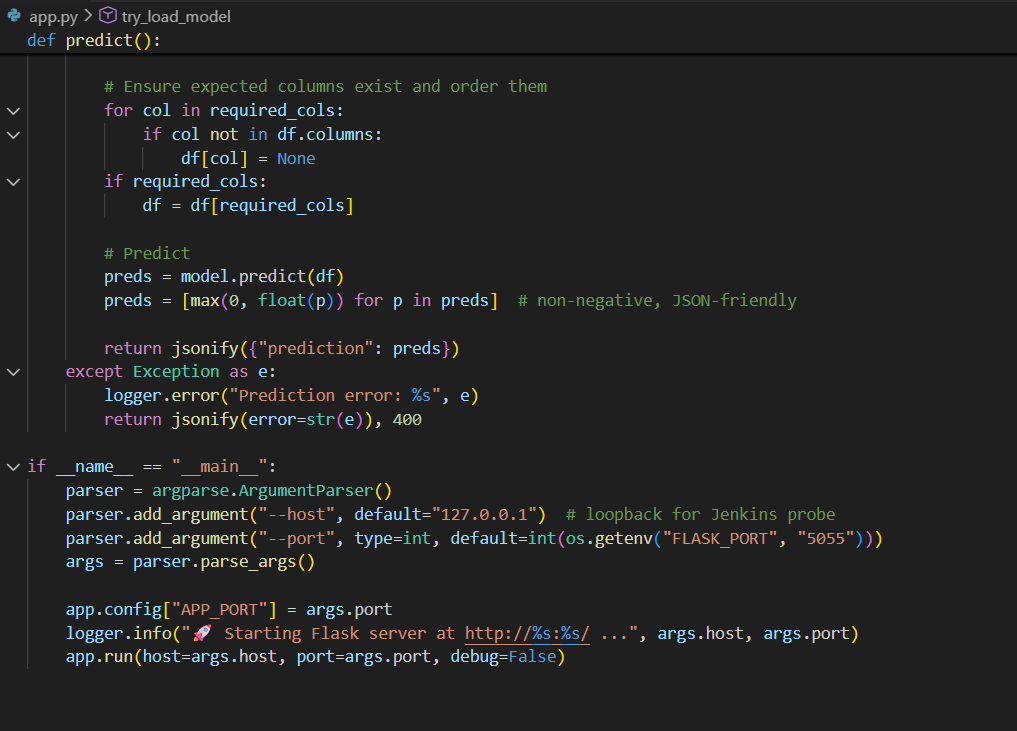
The prediction endpoint handles preprocessing tasks such as date feature extraction and missing column alignment. Predictions are constrained to non-negative values to maintain business logic consistency.

This ensures that the API can handle real-world input variations while maintaining reliable outputs.

**Code - app.py**

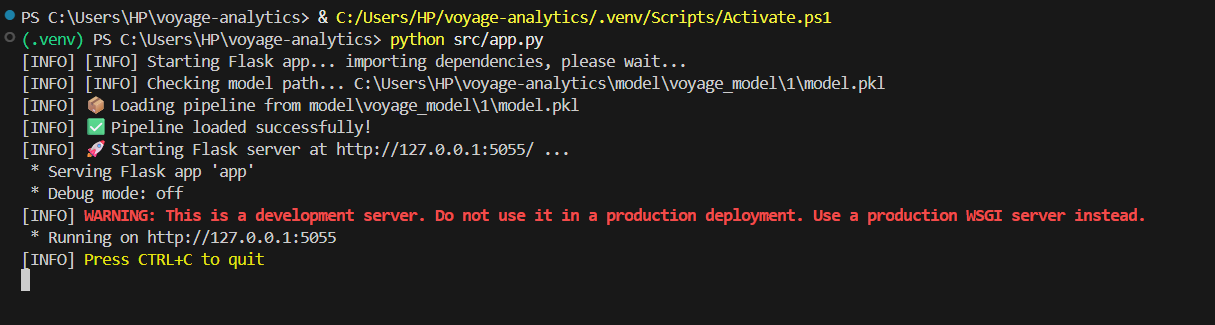




**Run:**

**python src/app.py**



**Code Explanation:**

**🔹 API Script (app.py)**

* Flask App Initialization  
  Creates a Flask application and sets up structured logging for observability.
* Model Loading  
  Loads the trained model asynchronously at startup to avoid blocking the API. Handles missing or failed loads gracefully.
* Metadata Handling  
  Reads columns.json to ensure incoming prediction requests match the model’s expected input format.
* Health Endpoint (/health)  
  Provides API and model status for monitoring and CI/CD readiness checks.
* Prediction Endpoint (/predict)  
  Accepts JSON input, performs date feature extraction, aligns input columns, and returns non-negative flight price predictions.
* Production Readiness  
  Includes error handling, schema validation, and configurable host/port settings for deployment and automation.

**Execution Summary**

The regression model training was executed using the train\_regression.py script by providing the users, flights, and hotels datasets as input. The datasets were successfully loaded and merged into a single dataframe containing 271,888 records and 21 features.

During feature preparation, identifier columns such as travel codes and user codes were removed to avoid data leakage. The final training dataset consisted of 6 numerical and 9 categorical features. A Scikit-learn pipeline was used to combine data preprocessing and a Random Forest Regressor into a single workflow.

Model training was tracked using MLflow, where model parameters, performance metrics, and artifacts were logged. The trained model achieved a high R² score of 0.9947, indicating strong predictive performance. The model was versioned and registered in the MLflow Model Registry, and the trained pipeline was saved locally along with feature metadata.

Following training, the Flask application (app.py) was executed. The API successfully loaded the trained model and started running on a local server. This API enables real-time flight price predictions through a REST endpoint and confirms the model is ready for deployment and further scaling.

**Testing Prediction Using Postman**

**What is Postman?**

Postman is an API testing tool that allows users to send HTTP requests to REST APIs and view their responses. It is commonly used to validate API functionality without writing additional client-side code. In this project, Postman is used to test the flight price prediction API by simulating real-world request scenarios.

**Postman Collection**

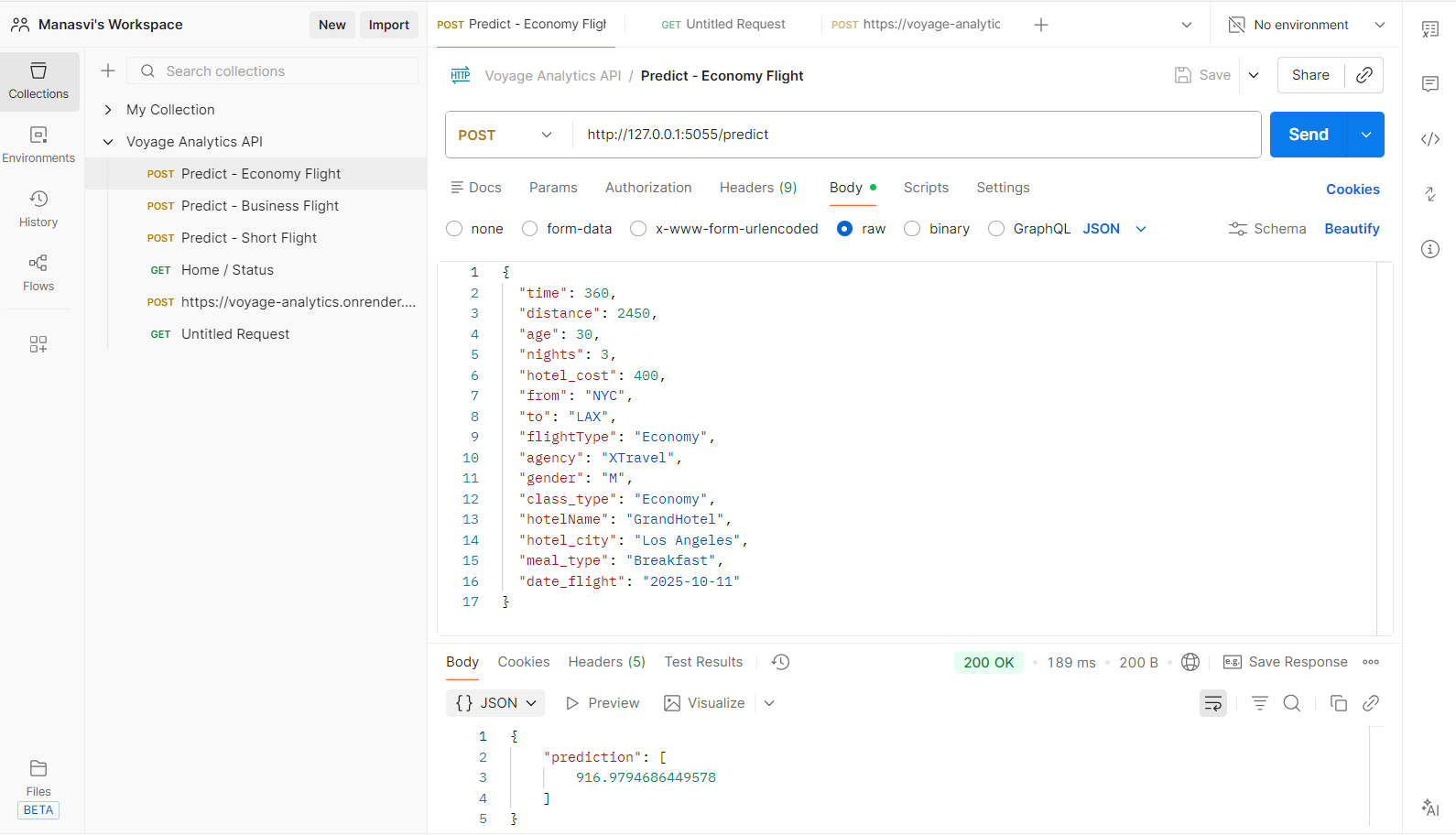
A Postman collection is created to group multiple API requests related to the Voyage Analytics project. This collection contains predefined requests for testing different flight scenarios such as economy flights, business flights, and short-distance flights, along with a basic API status check.

Each prediction request uses the **POST** method and sends input data in **JSON format** to the /predict endpoint. The JSON body includes both numerical features (such as flight duration, distance, age, hotel stay details) and categorical features (such as origin, destination, flight type, agency, and meal type). The flight date is also provided to allow the API to extract date-related features.

A **GET** request is included to verify that the API is running and accessible.

**Steps to Run Prediction in Postman**

1. Open Postman and click **Import**.
2. Select **Raw Text**, paste the provided Postman collection JSON, then click **Continue** and **Import**.
3. Ensure the Flask API is running at http://127.0.0.1:5055.
4. Select any request such as **Predict – Economy Flight** or **Predict – Business Flight**.
5. Click **Send** to trigger the prediction.

****

**Output**

On successful execution, the API returns a JSON response containing the **predicted flight price**.

Predict - Economy Flight

{

    "prediction": [

        916.9794686449578

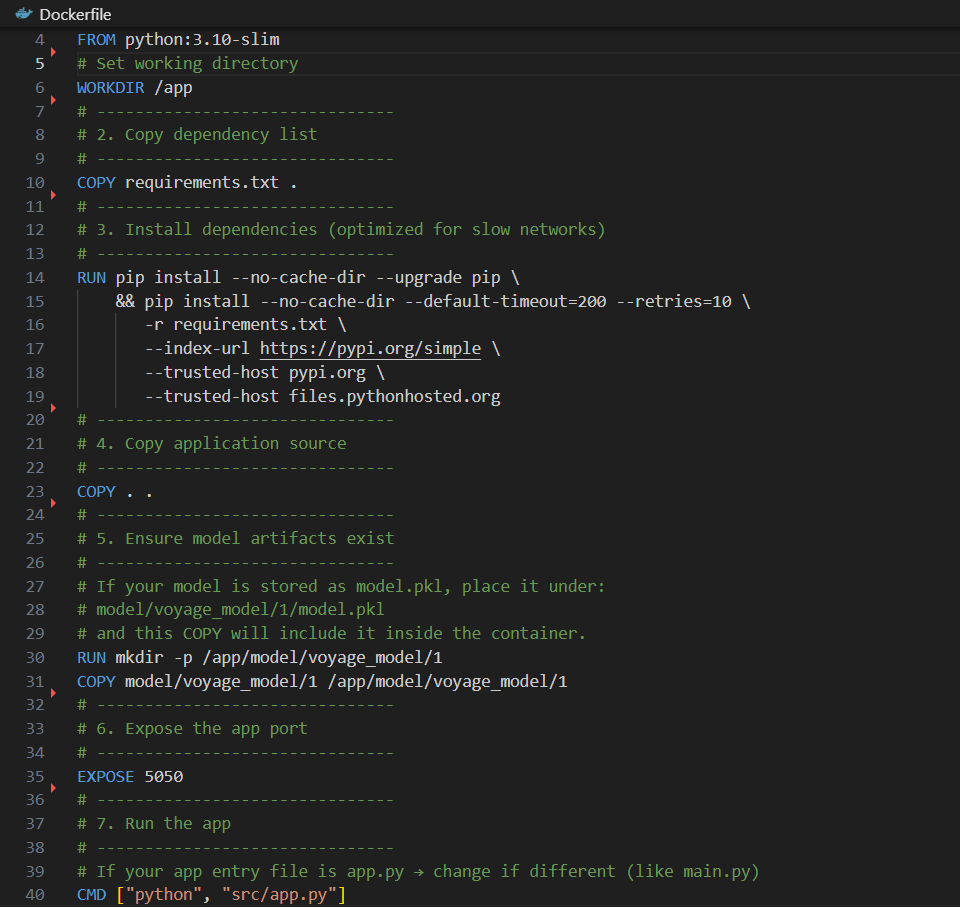
    ]

}

**Docker & Containerization**

**Dockerizing the Application**

Docker is a containerization platform that allows us to package an application along with its dependencies, libraries, and configuration into a single portable unit called a container. This ensures the application runs consistently across different environments such as development, testing, and production.

DockerFIle:

**Dockerfile Explanation**

* **Base Image**  
  FROM python:3.10-slim  
  Uses a lightweight Python image to reduce image size and improve performance.
* **Working Directory**  
  WORKDIR /app  
  Sets the default directory inside the container where the application will run.
* **Copy Dependencies**  
  COPY requirements.txt .  
  Copies the dependency list into the container.
* **Install Dependencies**  
  Installs Python packages with retries and extended timeout to handle slow networks efficiently.
* **Copy Application Code**  
  COPY . .  
  Copies the complete application source code into the container.
* **Model Artifacts Setup**  
  Ensures the trained machine learning model is available inside the container at the expected path.
* **Expose Port**  
  EXPOSE 5050  
  Exposes port 5050 for external access to the Flask API.
* **Run the Application**  
  CMD ["python", "src/app.py"]  
  Starts the Flask application when the container runs.

**Docker Commands Explanation**

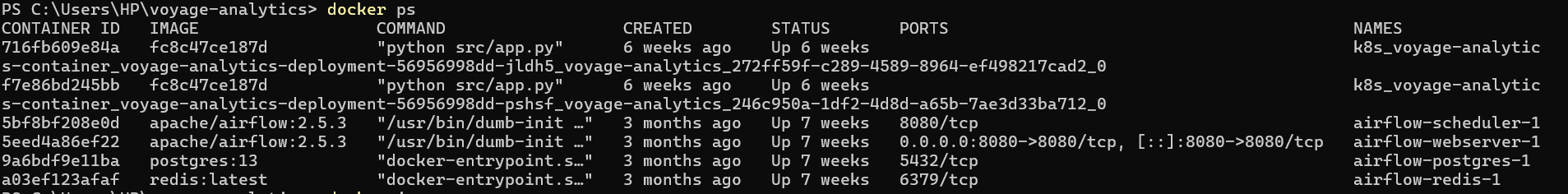
**1) Build Docker Image**

**docker build -t voyage-analytics-app .**

Builds a Docker image named voyage-analytics-app using the current project directory.

**2) List Running Containers**

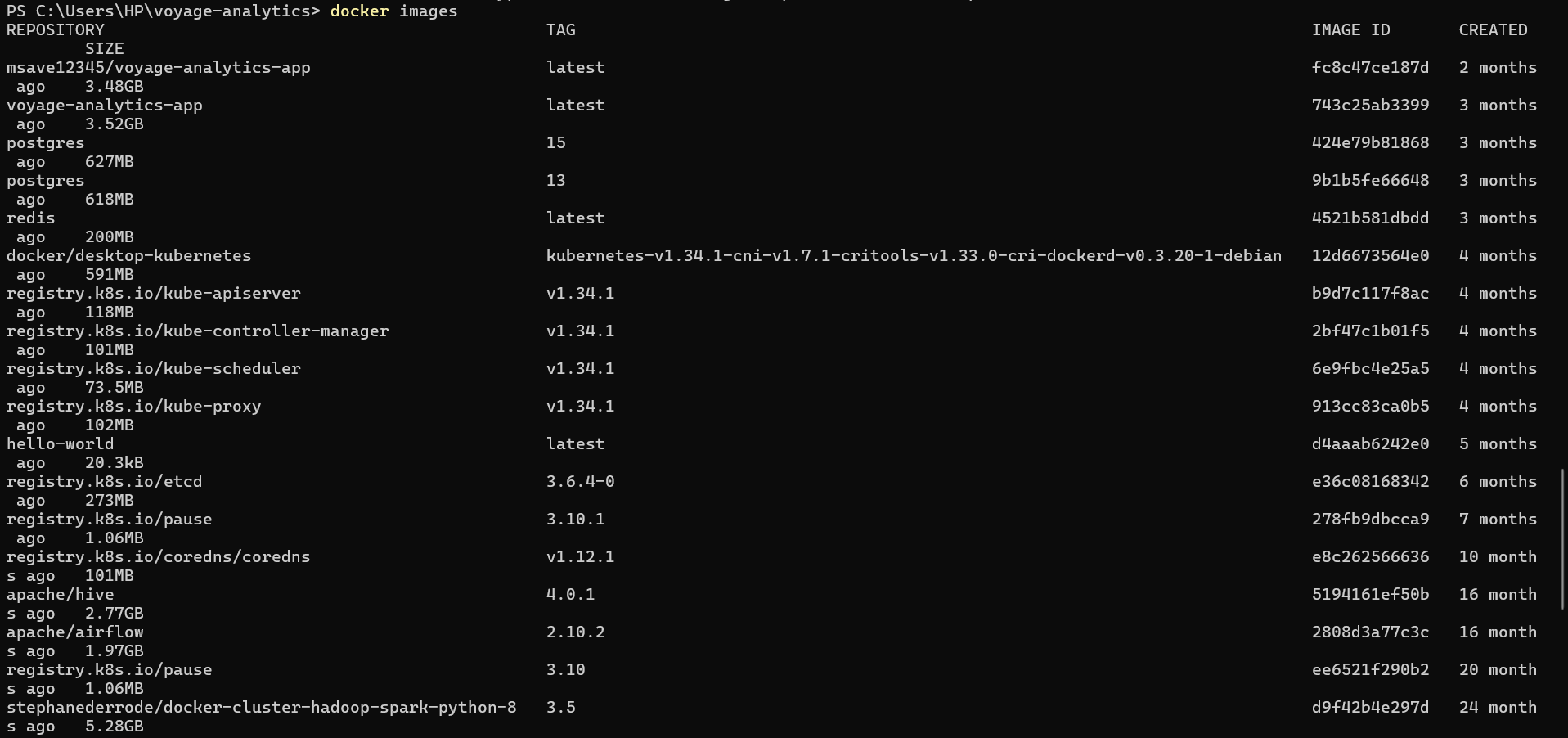
**docker ps**

Displays all currently running Docker containers.

**3) Verify Image Creation**

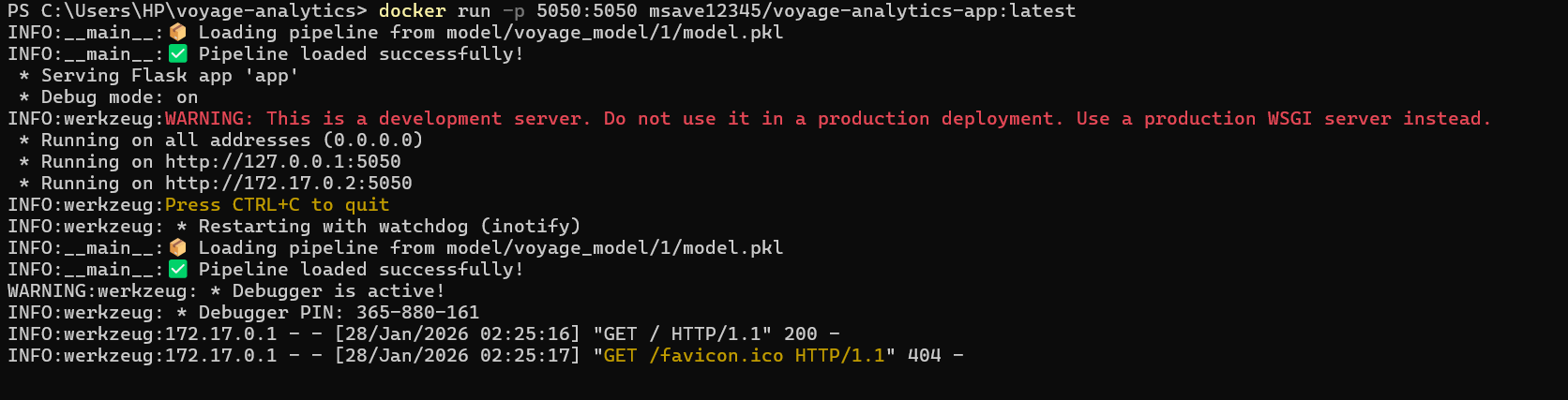
**docker images**

Lists all available Docker images on the system.



**4) Run the Container**

**docker run -p 5050:5050 voyage-analytics-app**

Runs the application and maps container port 5050 to the local machine’s port 5050.

 **Docker Container Running**

* The container named **thirsty\_burnell** is currently in **Running** state.
* The Docker image used is **msave12345/voyage-analytics-app:latest**.
* Port **5050** of the container is mapped to **5050** on the host machine, allowing external access to the Flask API.

**Container Logs Explanation**

* The application successfully loads the trained machine learning pipeline from:

**model/voyage\_model/1/model.pkl**

* The Flask application starts without errors and confirms:

Pipeline loaded successfully!

* The API is running on:
  + http://127.0.0.1:5050
  + http://172.17.0.2:5050 (internal Docker network)
* HTTP GET / requests returning **200** indicate that the API is reachable and responding correctly.

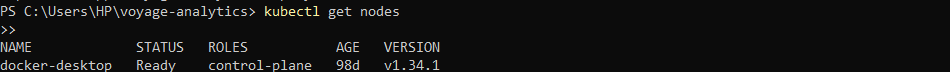
**Kubernetes**

Kubernetes is a container orchestration platform used to deploy, manage, scale, and monitor containerized applications. While Docker runs a single container, Kubernetes manages multiple containers across clusters, ensuring high availability, fault tolerance, and scalability. It automatically handles container restarts, load balancing, and scaling based on demand.

**Enabling Kubernetes in Docker Desktop**

To use Kubernetes locally, Docker Desktop is launched and Kubernetes is enabled from the Docker Desktop settings. Once enabled, Docker Desktop initializes a local Kubernetes cluster.

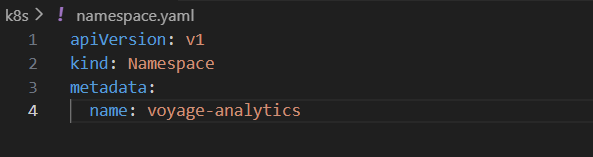
To verify that Kubernetes is running, the following command is executed in PowerShell:

kubectl get nodes  
  


**Kubernetes Configuration Files**

Three YAML configuration files are created to deploy the application in Kubernetes:

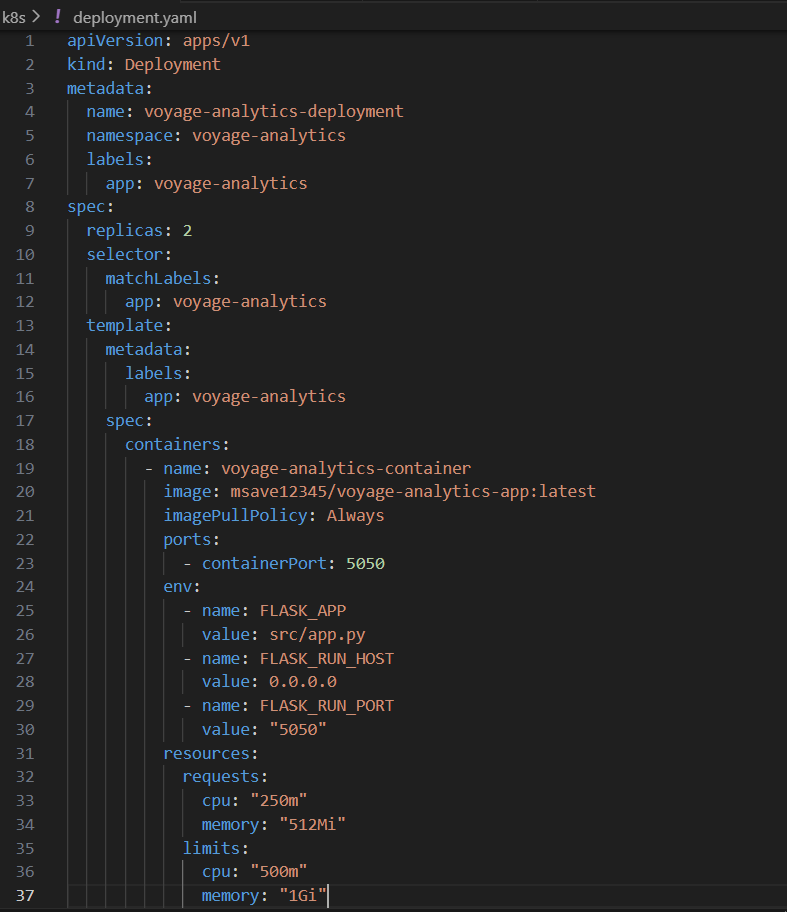
1. **Namespace (namespace.yaml)**  
   A namespace is used to logically isolate the Voyage Analytics application from other workloads. This helps in better organization and management of Kubernetes resources.



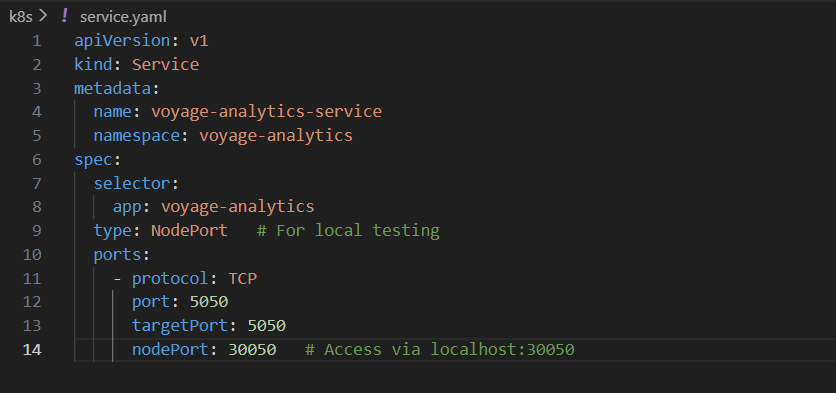
1. **Deployment (deployment.yaml)**  
   The deployment file defines:

* The Docker image to use
* Number of replicas (pods)
* Container port
* Restart and scaling behavior

Kubernetes ensures the specified number of application pods are always running.



1. **Service (service.yaml)**  
   The service file exposes the application to external traffic. It provides a stable network endpoint and load balances traffic across all running pods.

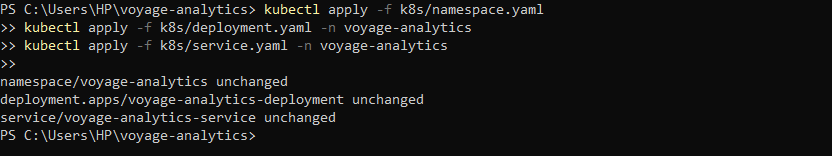


Deploying the Application to Kubernetes

kubectl apply -f k8s/namespace.yaml

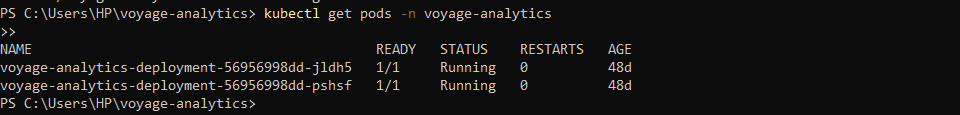
kubectl apply -f k8s/deployment.yaml -n voyage-analytics

kubectl apply -f k8s/service.yaml -n voyage-analytics



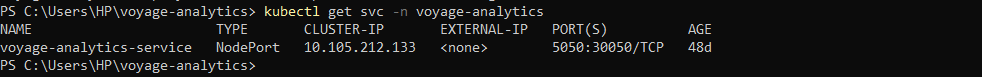
Verifying the Deployment

kubectl get pods -n voyage-analytics



To check the service status:

kubectl get svc -n voyage-analytics



To verify all Kubernetes resources:

kubectl get all -n voyage-analytics



Successful output confirms that the application is fully deployed and running inside the Kubernetes cluster.

**Directed Acyclic Graph**

A DAG stands for **Directed Acyclic Graph**. It is a way of defining a workflow where tasks are executed in a specific order.

* **Directed** means tasks flow in one direction
* **Acyclic** means there are no loops
* **Graph** represents tasks and their dependencies

In data engineering and MLOps, DAGs are commonly used to orchestrate pipelines such as data ingestion, preprocessing, model training, and deployment. Each task runs only after its dependent task has completed successfully.

DAGs are most commonly implemented using tools like **Apache Airflow**, where each workflow is defined as code.

**In the Voyage Analytics project, DAGs are used to:**

* Automate the end-to-end ML workflow
* Ensure tasks run in the correct sequence
* Make the pipeline repeatable and reliable
* Reduce manual intervention

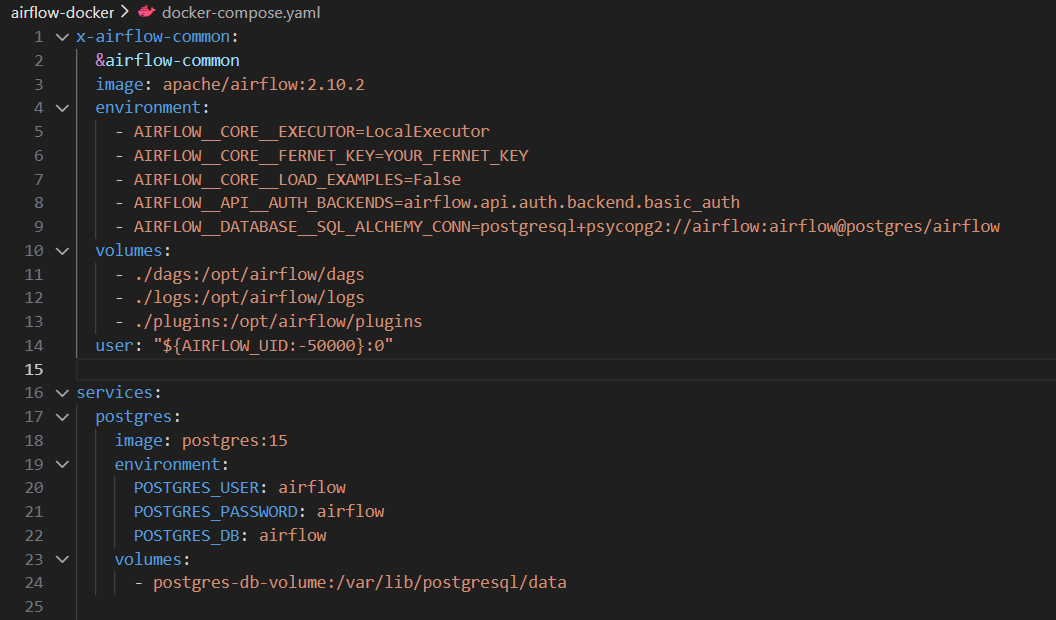
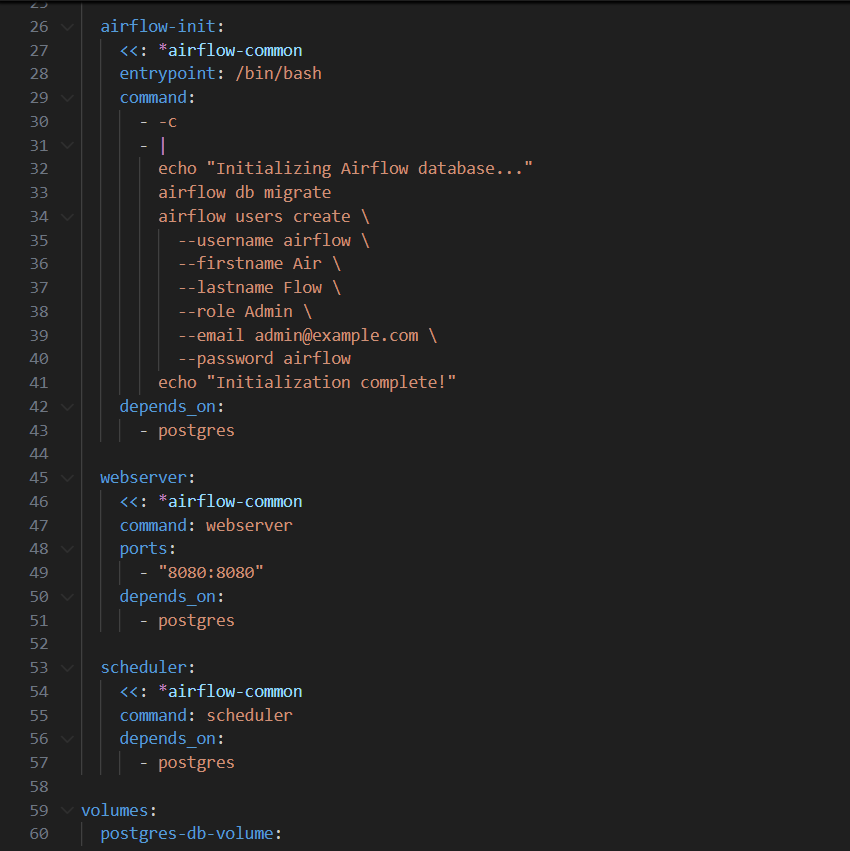
**Docker Compose for DAG Setup**

To run DAG-related services locally, a docker-compose.yaml file is created.  
Docker Compose allows multiple services (such as Airflow webserver, scheduler, database, and worker) to be defined and started together using a single command.

Using Docker Compose ensures:

* Consistent environment setup
* Easy startup and shutdown of services
* Simplified local testing of DAGs

The docker-compose.yaml file defines all required services, networks, ports, and dependencies needed to run DAG orchestration smoothly.

This Docker Compose setup creates a fully functional Airflow environment where DAGs can be developed, tested, and executed locally. It enables reliable orchestration of machine learning workflows in a reproducible and scalable manner.

**Running Apache Airflow**

After creating the Docker Compose configuration, the next step is to start the Airflow services.

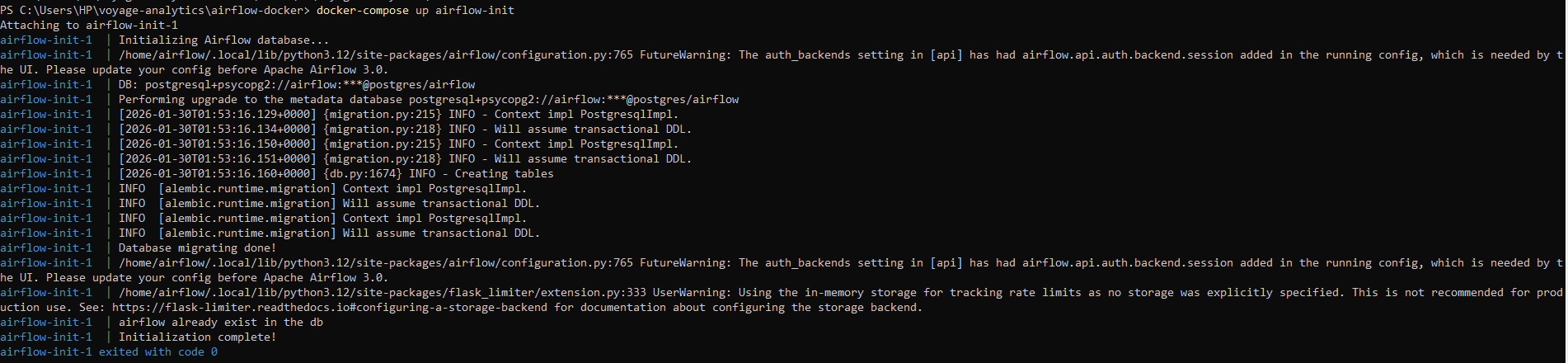
First, initialize the Airflow environment by running:

**docker-compose up airflow-init**

This command performs the following actions:

* Initializes the Airflow metadata database
* Applies database migrations
* Creates an admin user for accessing the Airflow UI

This step needs to be executed **only once** during the initial setup.

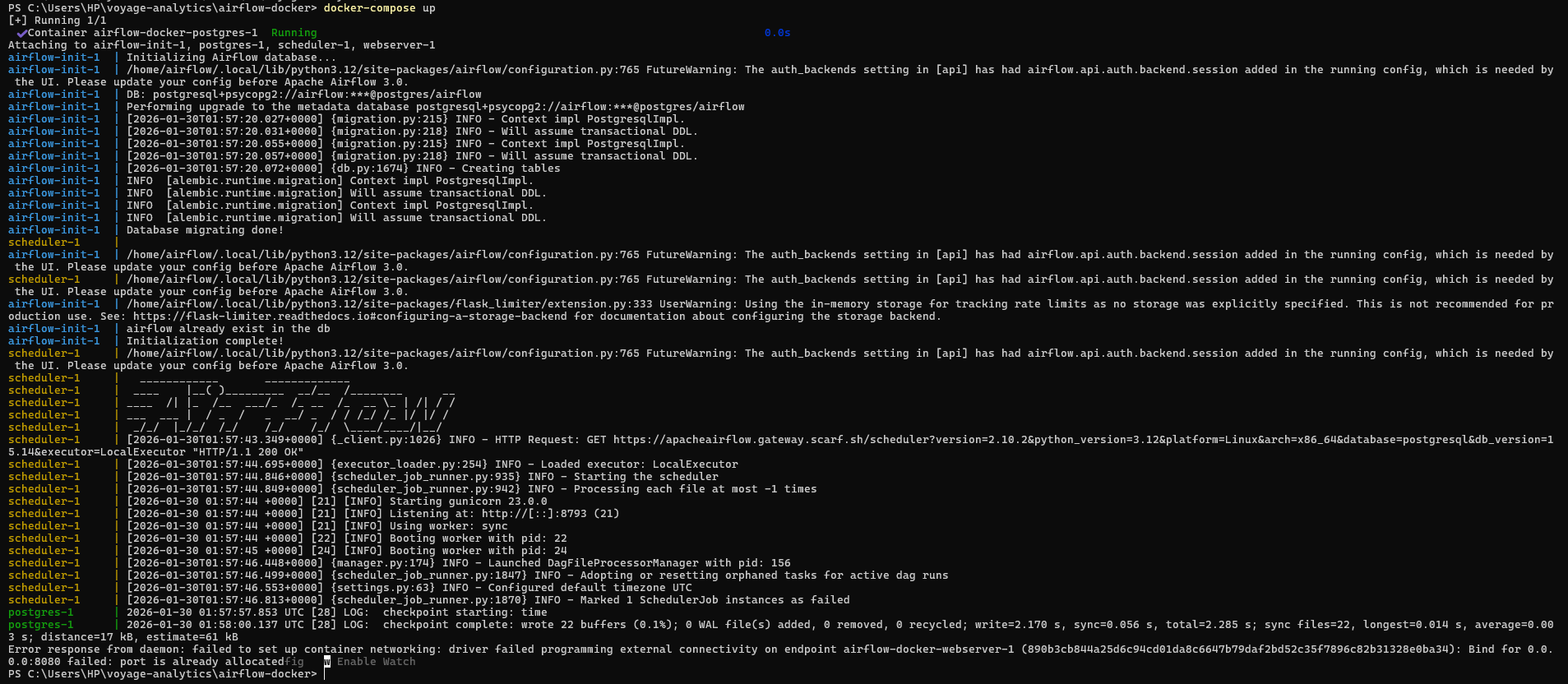


**Next, start all Airflow services using:**

docker-compose up

This command starts:

* **PostgreSQL** (metadata database)
* **Airflow Webserver** (UI access)
* **Airflow Scheduler** (DAG scheduling and execution)



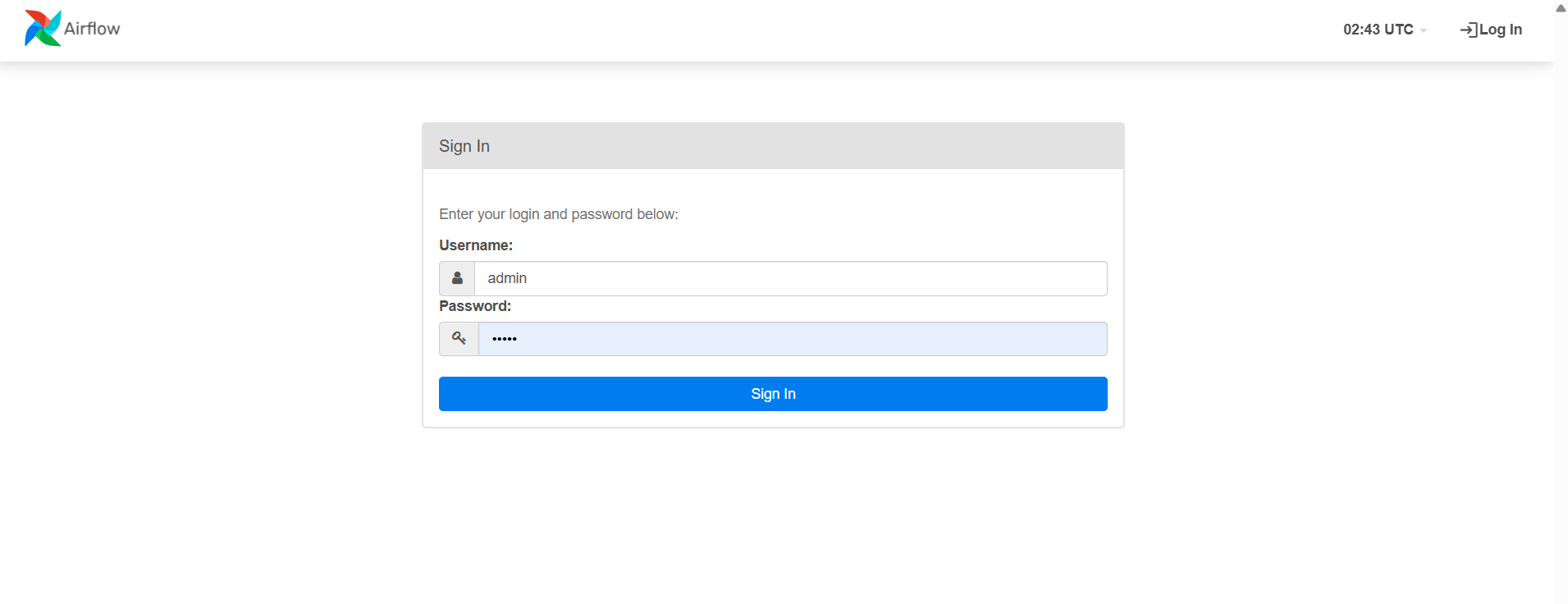
**Accessing the Airflow UI**

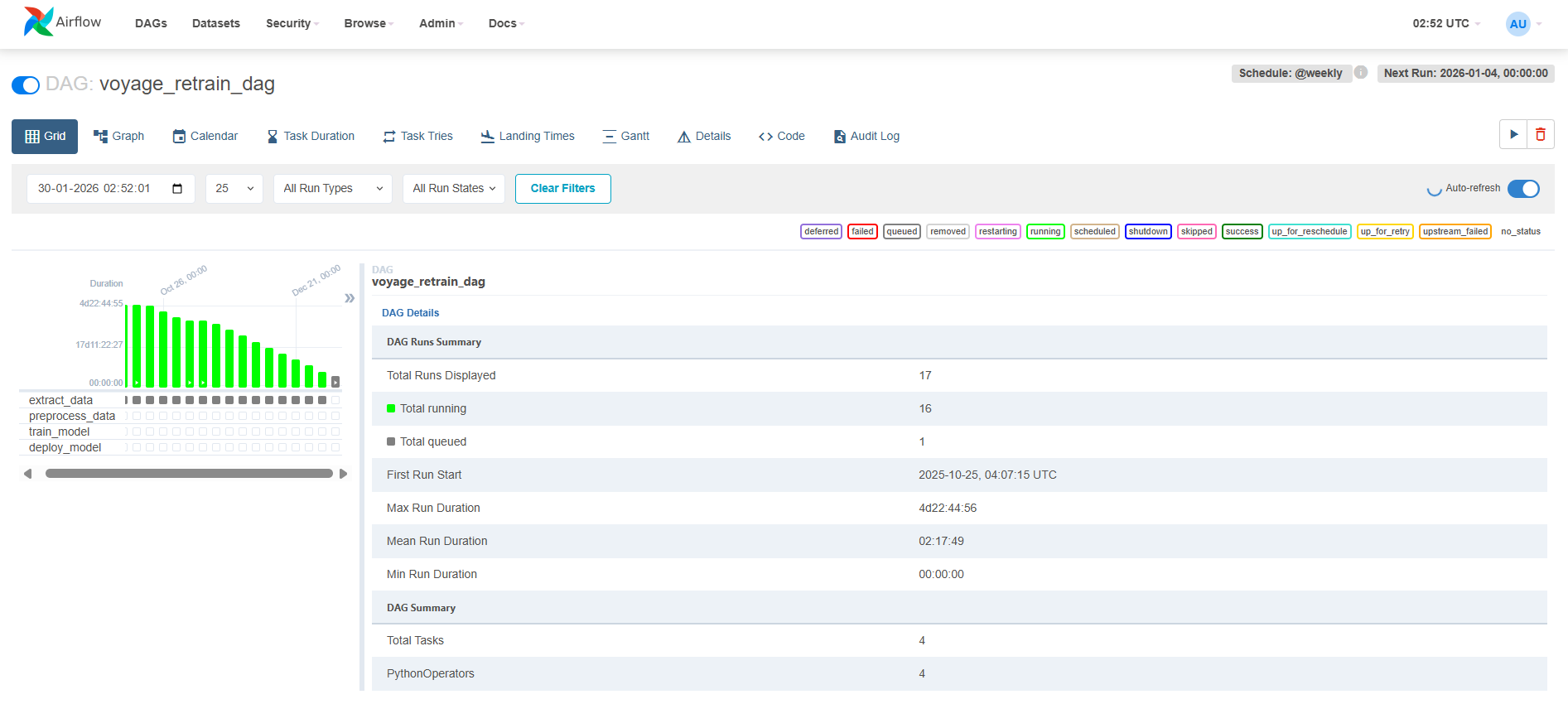
Once all services are running, open the Airflow web interface in a browser:

[**http://localhost:8080**](http://localhost:8080)

From the UI, you can:

* View available DAGs
* Trigger workflows manually
* Monitor task execution status
* Inspect logs and execution history



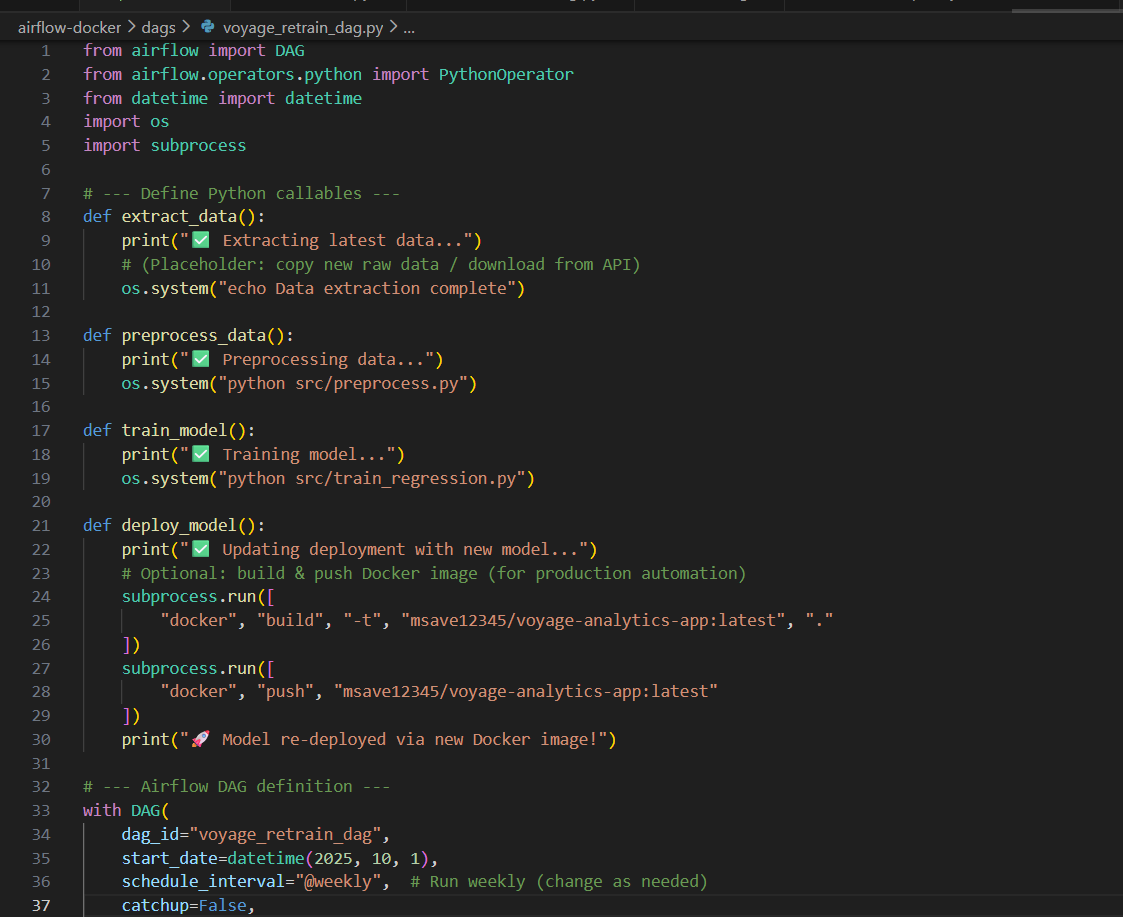
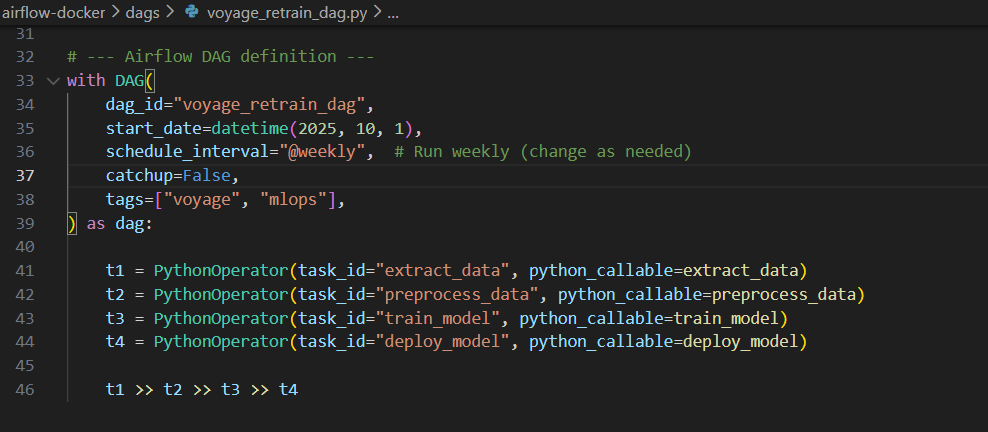


**Creating an Apache Airflow DAG**

To automate the machine learning workflow, a Directed Acyclic Graph (DAG) is created using Apache Airflow. The DAG defines a structured and repeatable pipeline for the Voyage Analytics project.

The file **voyage\_retrain\_dag.py** is placed inside the dags/ directory so that Ai

rflow can automatically detect and register it.

**DAG Code Explanation**

* The DAG imports core Airflow components such as DAG and PythonOperator.
* Three Python functions are defined to represent different pipeline stages:
  + **load\_data** – simulates loading and preparing datasets
  + **train\_model** – represents model training logic
  + **deploy\_model** – simulates model deployment

Each function is connected to a **PythonOperator**, allowing Airflow to execute Python code as a task.

**DAG Configuration**

* **DAG Name:** voyage\_analytics\_dag
* **Schedule:** Runs once every day (@daily)
* **Start Date:** October 10, 2025
* **Retries:** 1 retry with a delay of 1 minute
* **Catchup:** Disabled to avoid backfilling old runs

**Jenkins CI/CD Pipeline**

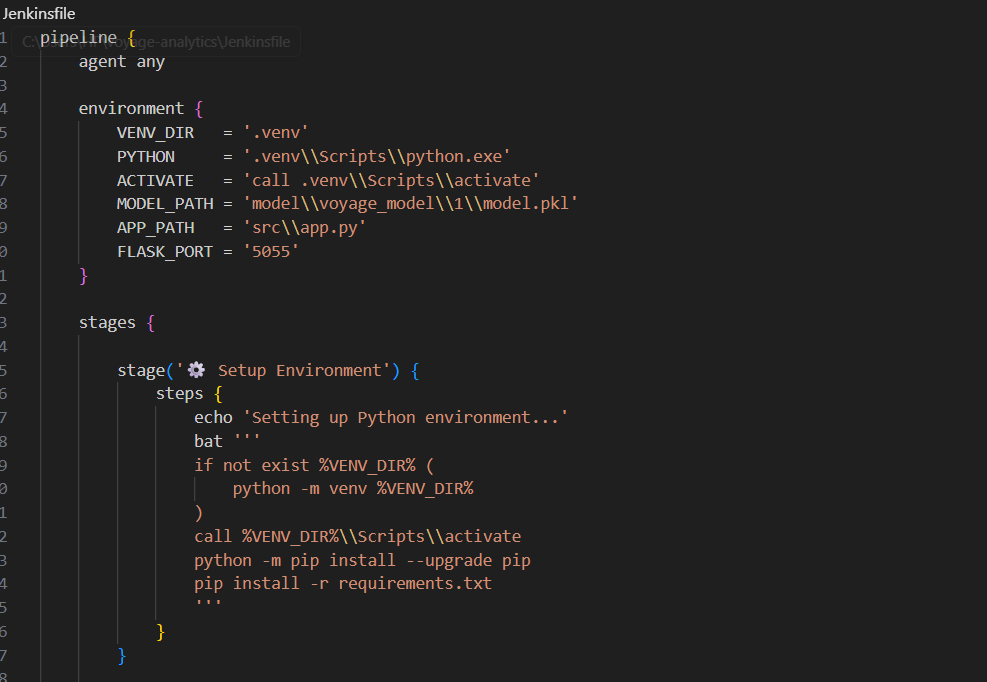
**What is Jenkins?**

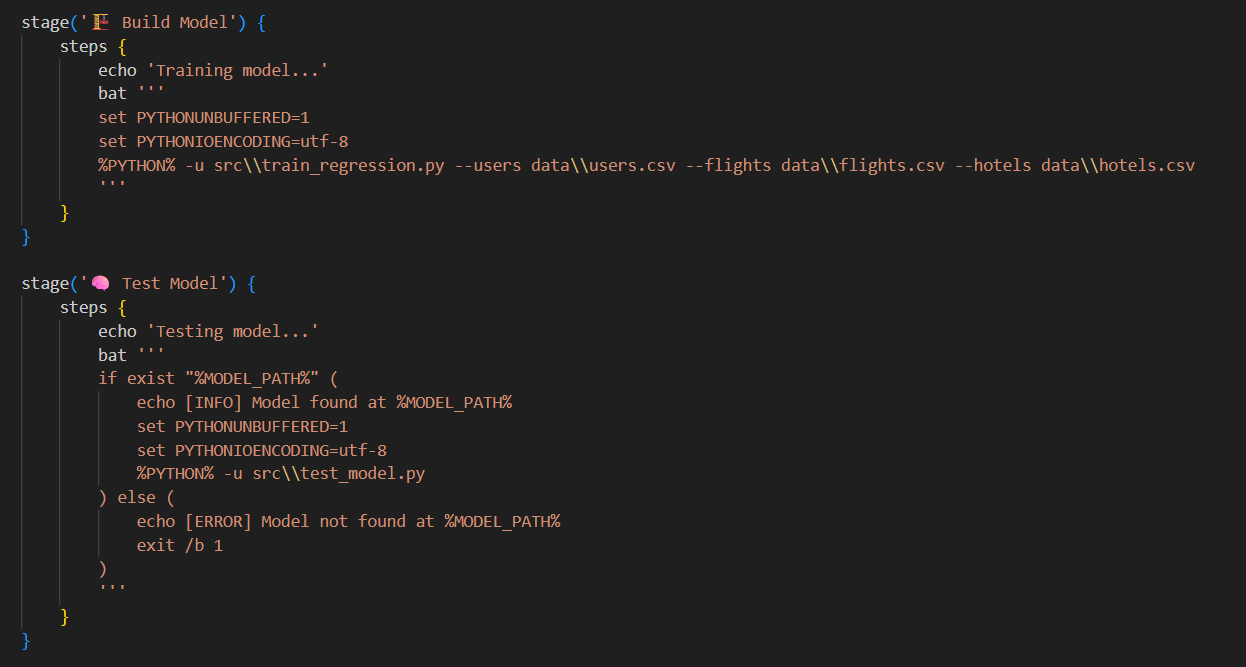
Jenkins is an open-source automation tool used for **Continuous Integration and Continuous Deployment (CI/CD)**. It automatically builds, tests, and deploys applications whenever new code is pushed to a repository, ensuring faster and reliable delivery.

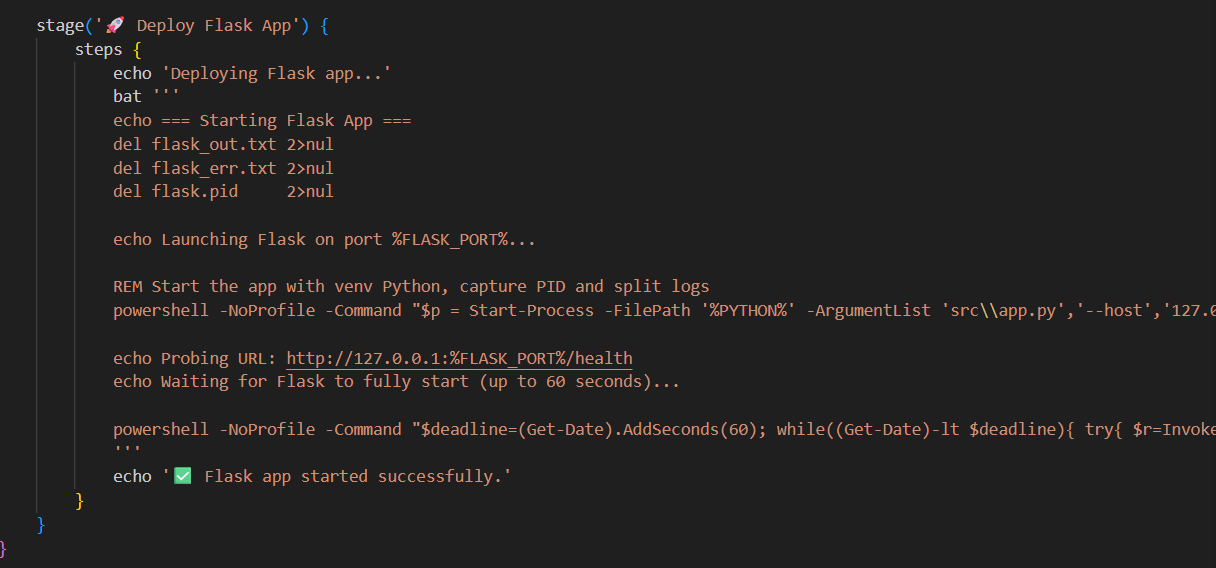
In this project, Jenkins is used to:

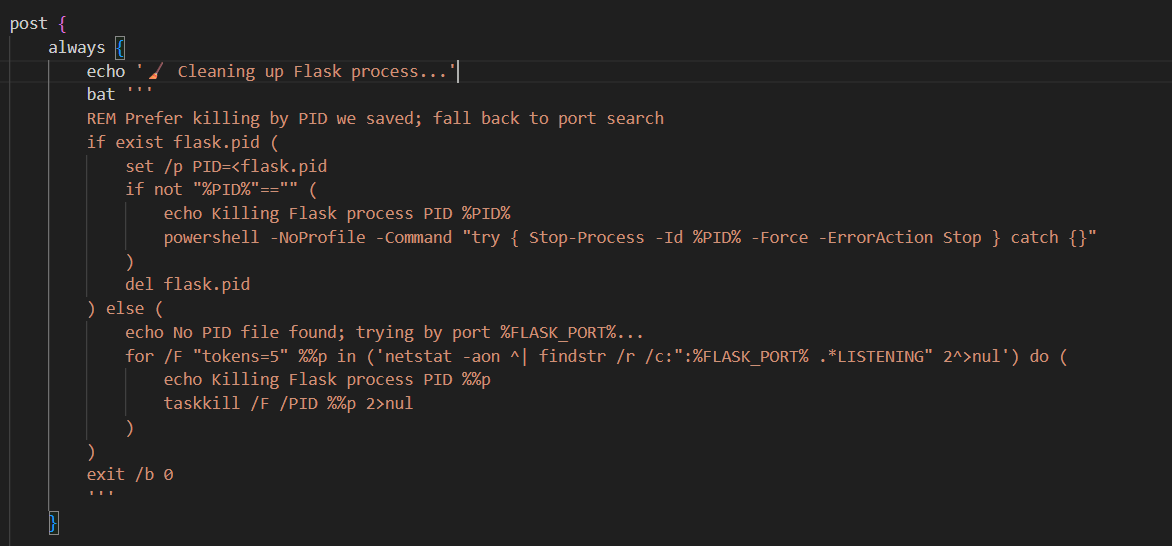
* Set up the Python environment
* Train the machine learning model
* Test the trained model
* Deploy the Flask API automatically

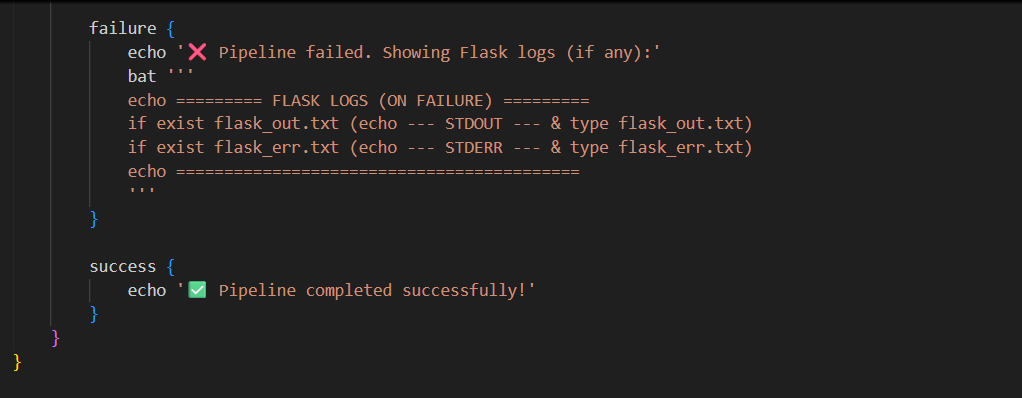
JenkinsFile:











**Pipeline Stages**

**1. Setup Environment**

* Creates a Python virtual environment if it does not exist
* Activates the virtual environment
* Installs all required dependencies from requirements.txt

**2. Build Model**

* Runs the model training script
* Generates the trained model file (model.pkl)
* Logs training output for monitoring

**3. Test Model**

* Verifies that the trained model exists
* Executes a test script to validate model loading and predictions
* Fails the pipeline if the model is missing

**4. Deploy Flask App**

* Starts the Flask API using the trained model
* Runs the application on the specified port
* Performs a health check to confirm the API is running successfully

**Post-Build Actions**

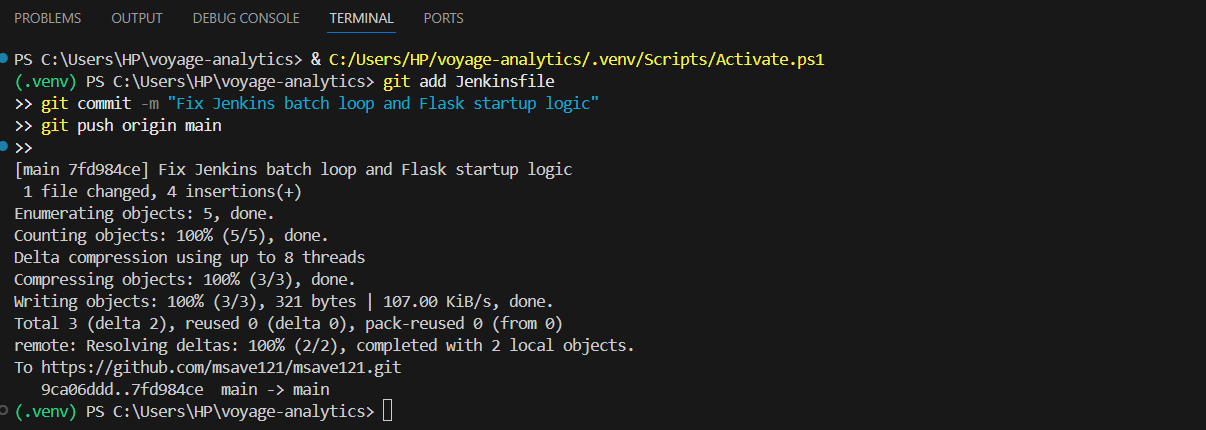
* **Always:**  
  Stops the Flask process and cleans up resources
* **Failure:**  
  Displays Flask application logs for debugging
* **Success:**  
  Confirms successful completion of the pipeline

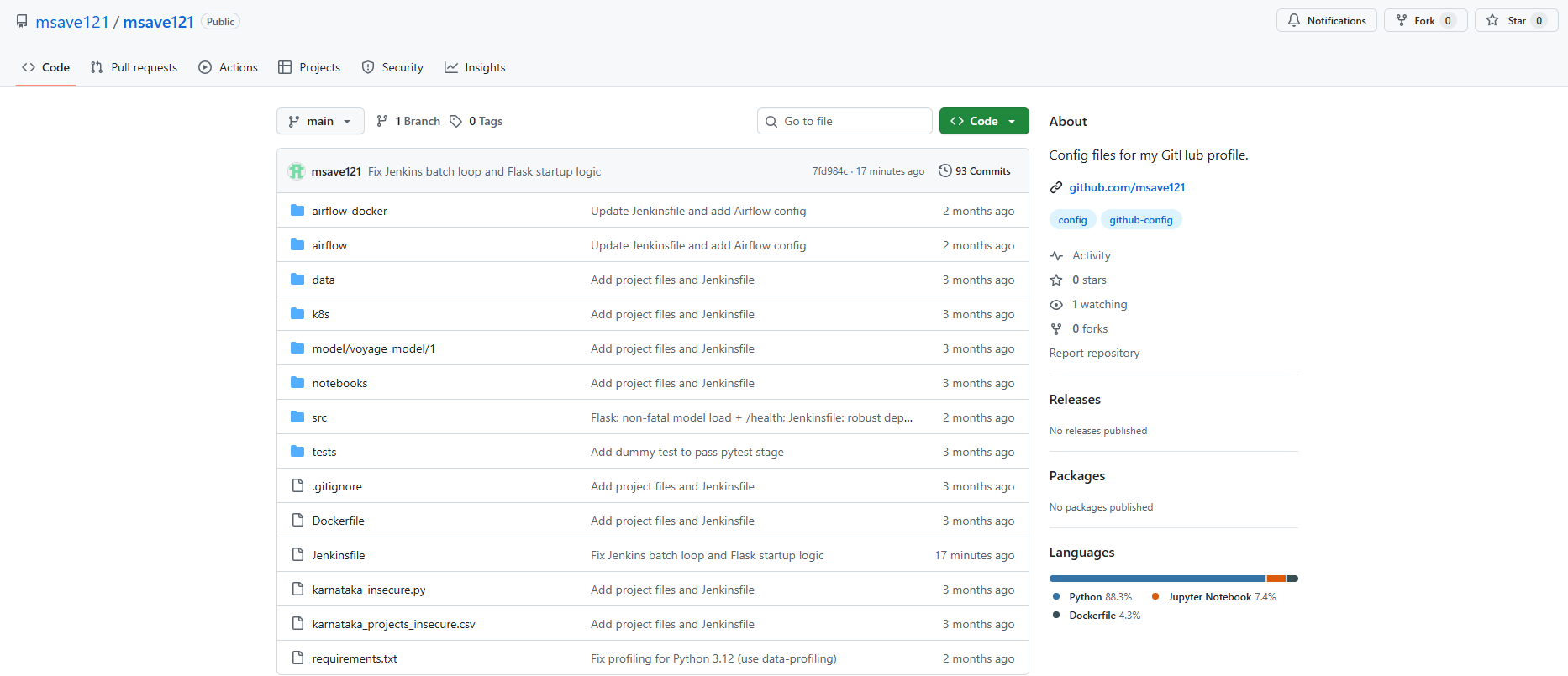
**Committing Jenkinsfile and Triggering the CI/CD Pipeline**

**The following commands are used:**

git add Jenkinsfile

git commit -m "Add Jenkins CI/CD pipeline"

git push origin main

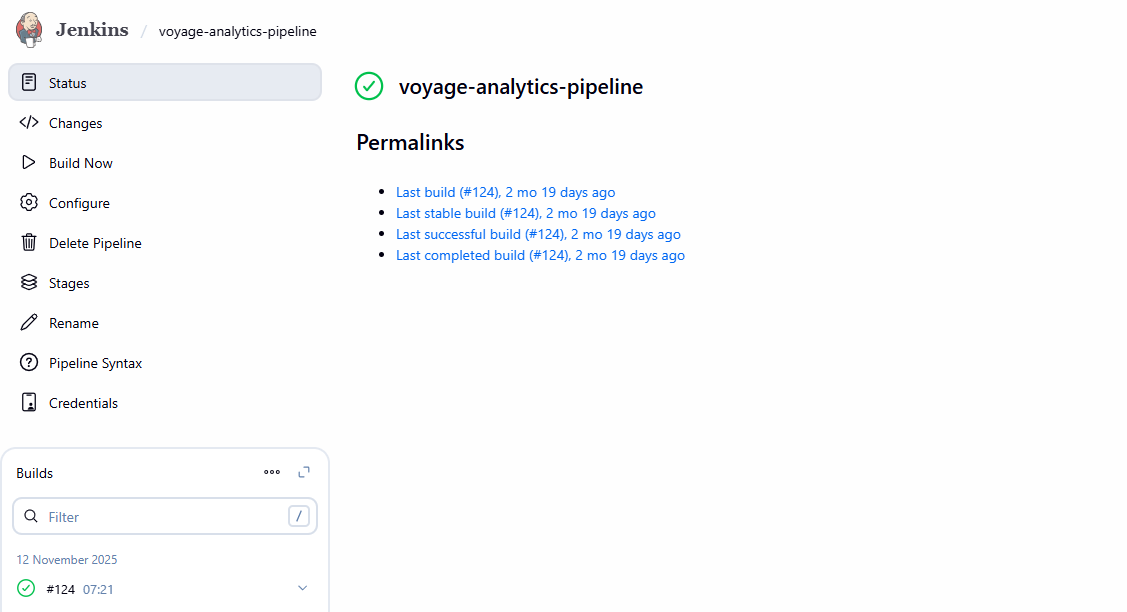
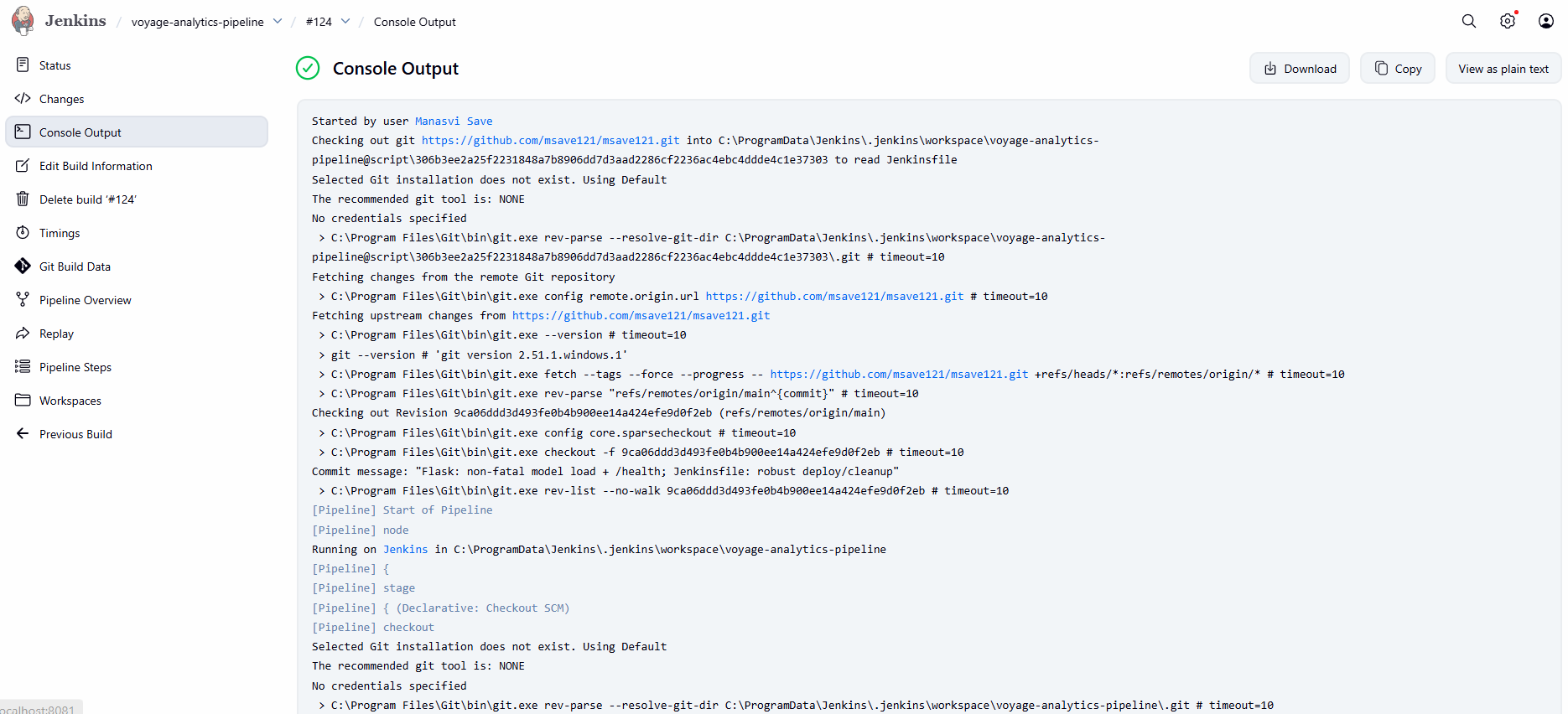
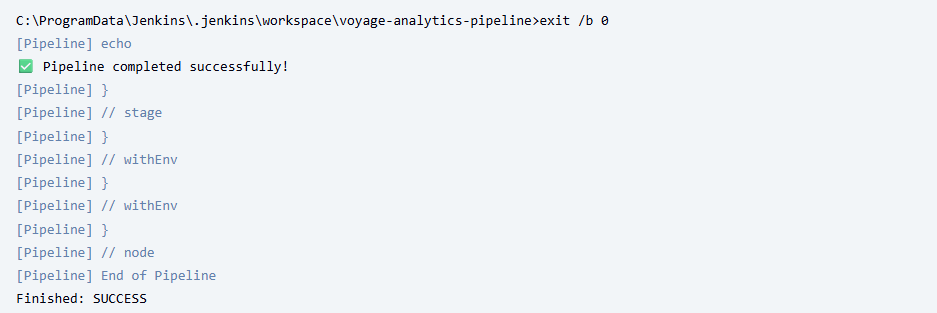
Once the Jenkinsfile is pushed, Jenkins automatically detects it from the repository.

Next, navigate to:

Jenkins Dashboard → Pipeline Job → Build Now

When the build is triggered, Jenkins reads the Jenkinsfile from the GitHub repository and executes each pipeline stage sequentially:

* Environment setup
* Model training
* Model testing
* Flask application deployment

**Console Output**

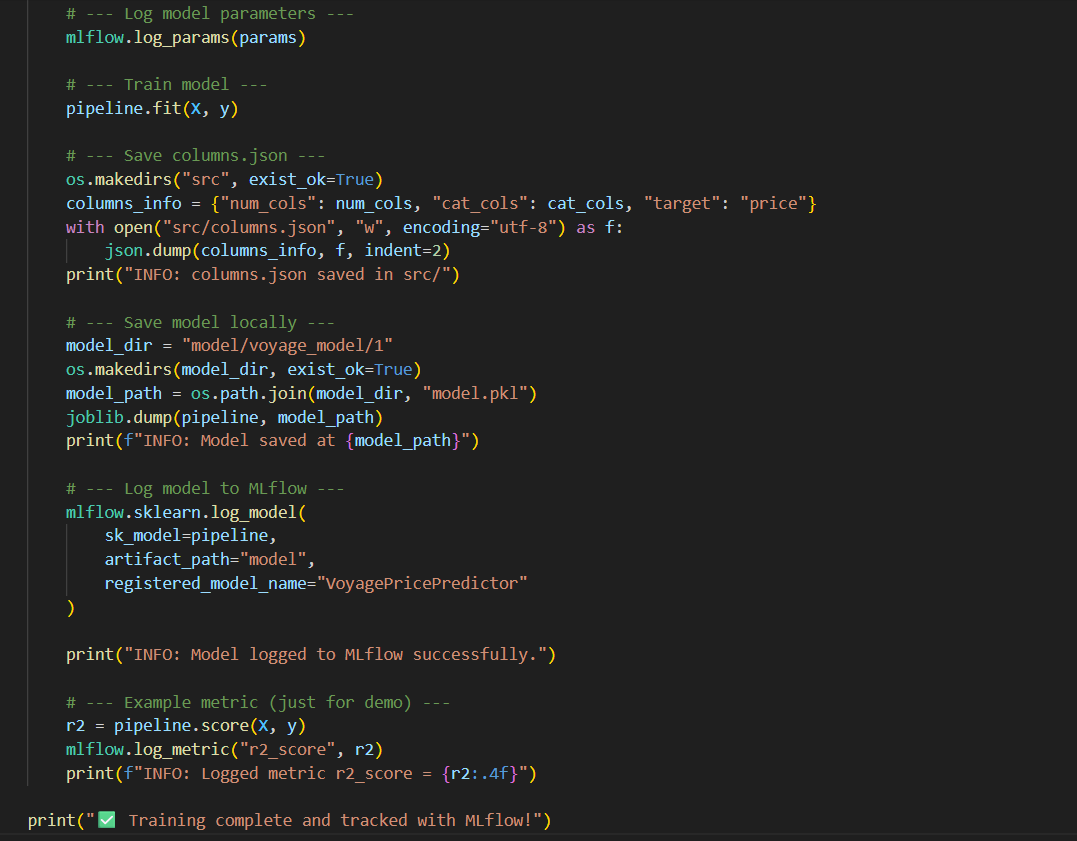
**Model Tracking with MLflow**

**MLflow** is an open-source platform used to track machine learning experiments, manage model versions, and store training metadata such as parameters, metrics, and artifacts.

In this project, MLflow is integrated into the training script to track:

* Model parameters
* Evaluation metrics
* Trained model artifacts

**MLflow Integration in Training Script**

****

During model training, MLflow is used to log important information:

* **Parameters Logging:**  
  Model hyperparameters such as number of trees and maximum depth are logged using mlflow.log\_params().
* **Model Training:**  
  The regression pipeline is trained on the prepared dataset.
* **Artifacts Storage:**  
  The trained model is saved locally and also logged to MLflow as an artifact.  
  Feature metadata is stored in columns.json for consistent inference.
* **Model Registration:**  
  The trained model is registered in the MLflow Model Registry under the name **VoyagePricePredictor**.
* **Metrics Logging:**  
  Model performance is evaluated using the R² score, which is logged using mlflow.log\_metric().

This setup enables systematic tracking and comparison of multiple training runs.

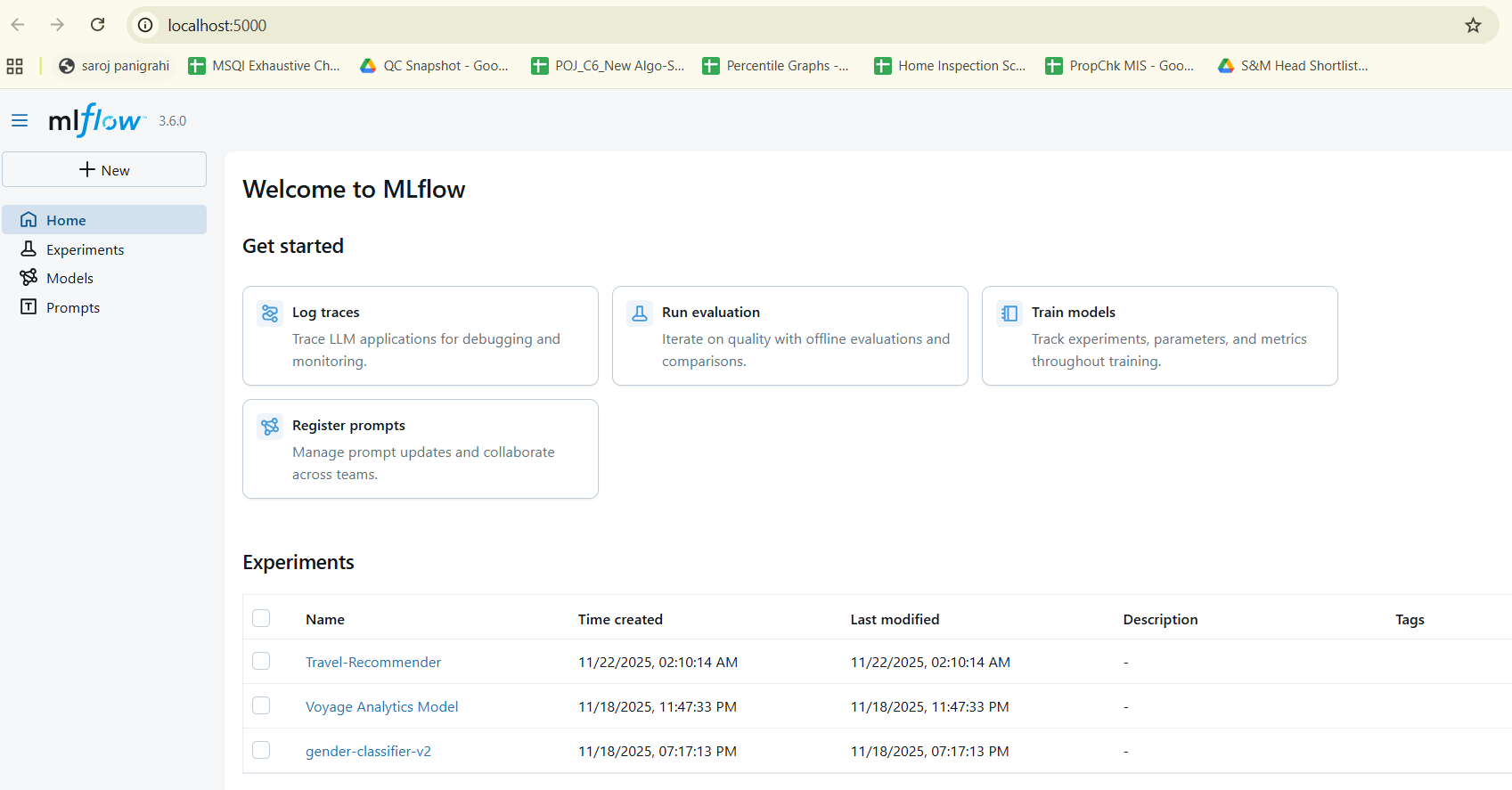
**Viewing Experiments in MLflow UI**

After training, the MLflow UI is started using:

**mlflow ui**

The UI is accessible at:

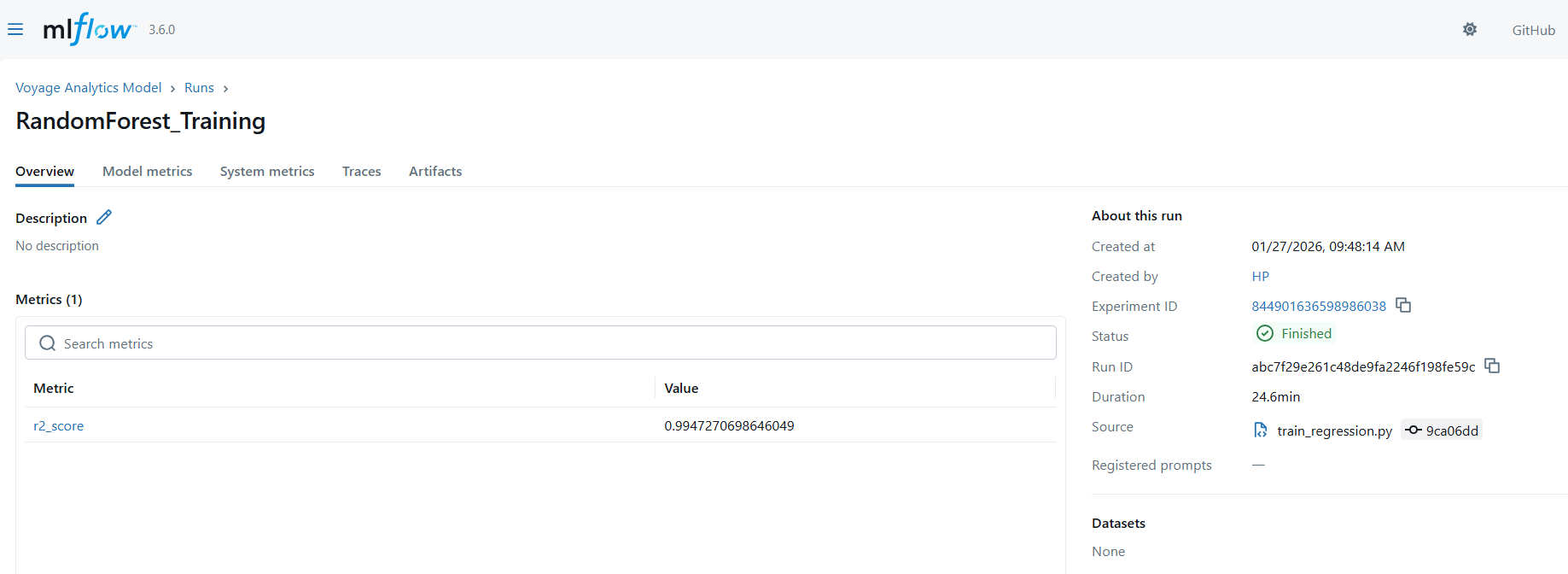
[**http://localhost:5000**](http://localhost:5000)

****

Under the experiment “Voyage\_Analytics\_Model”, new training runs appear automatically.  
Each run displays:

* **Parameters:** Model configuration and training details
* **Metrics:** Performance scores such as R²
* **Artifacts:** Saved models and related files

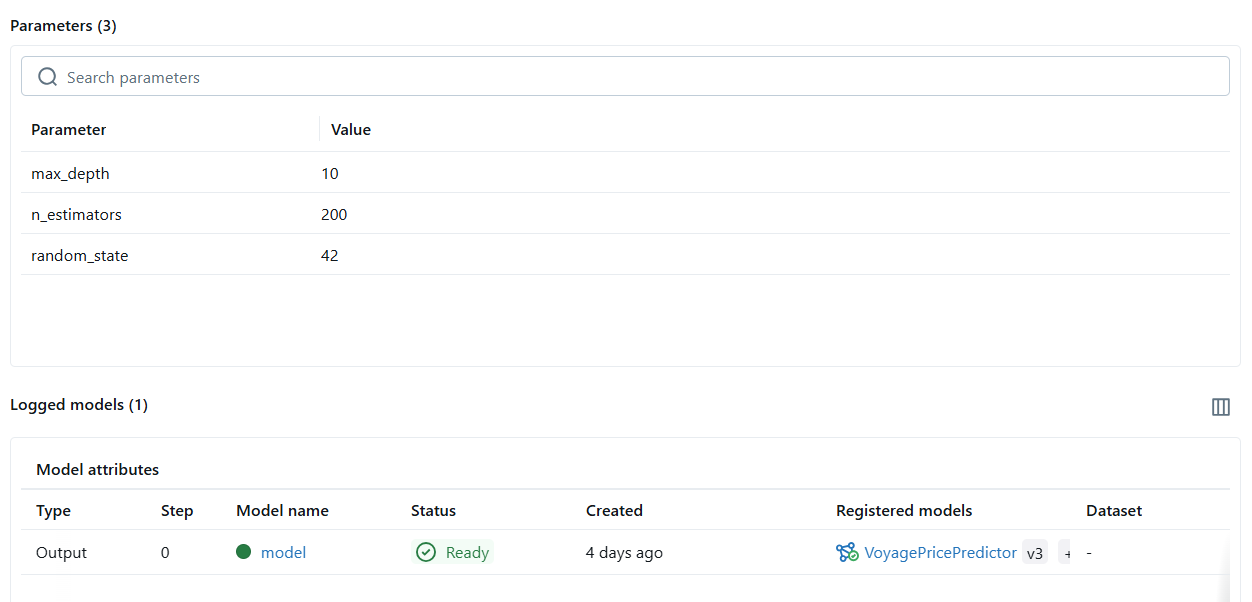
This ensures complete visibility and version control of the machine learning lifecycle.

****

Metric: **R² Score**

* **R² Score**: 0.9947

The R² score indicates how well the model explains the variance in the target variable (flight price).  
A value close to 1.0 means the model predictions are highly accurate.  
In this case, an R² score of 0.9947 shows that the model explains nearly 99.47% of the variability in flight prices, indicating excellent performance.

****

The following hyperparameters were used during training:

* **n\_estimators = 200**  
  Number of decision trees in the Random Forest model. A higher number improves stability and accuracy.
* **max\_depth = 10**  
  Limits the depth of each tree to prevent overfitting and improve generalization.
* **random\_state = 42**  
  Ensures reproducibility by generating the same results across multiple runs.

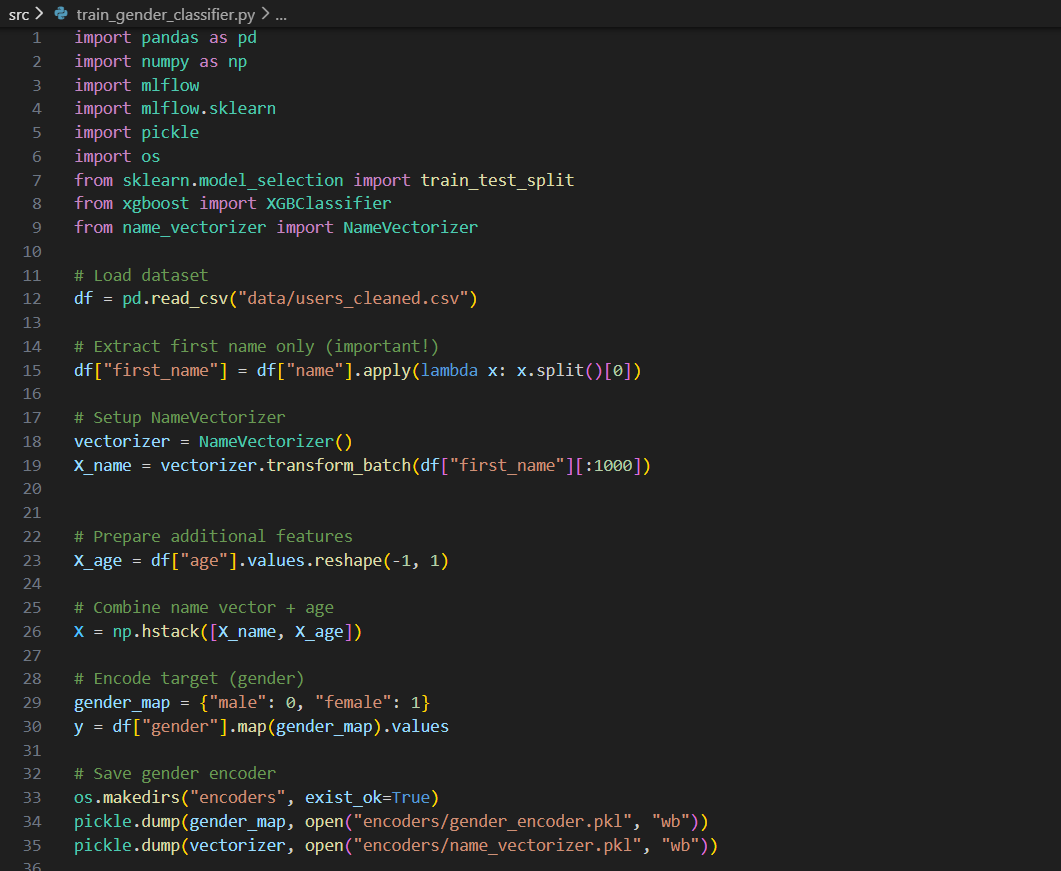
**Gender Classification Model**

This model predicts **gender (male / female)** using:

* **First name** (converted into numerical vectors using NameVectorizer)
* **Age** (as an additional numerical feature)

So the model learns **patterns in names + age** to classify gender.

**Train Gender Classifier:**

****

****

**Model & Training Explanation**

**Features used**

* **Name vector →** captures spelling patterns
* **Age →** helps improve prediction accuracy

These features are combined using:

X = np.hstack([X\_name, X\_age])

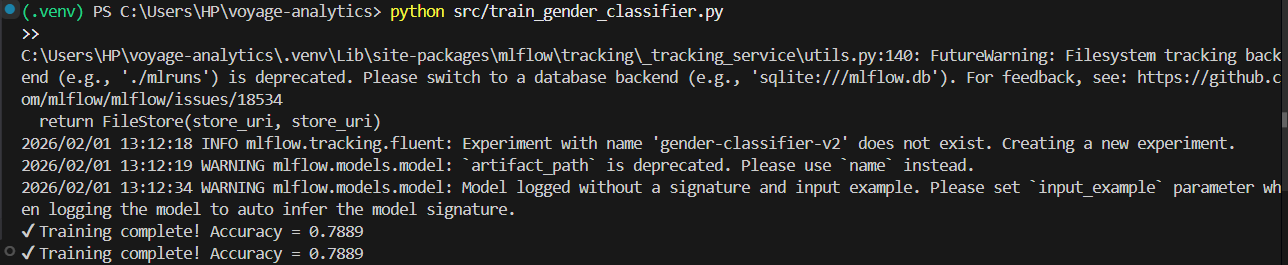
**Model used: XGBoost Classifier**

* n\_estimators = 50 → number of trees
* max\_depth = 6 → tree complexity
* learning\_rate = 0.1 → step size during learning

XGBoost is chosen because:

* It handles complex patterns well
* It performs strongly on tabular data
* It is fast and scalable

**Accuracy Result**

** Accuracy = 0.7889 (~79%)**

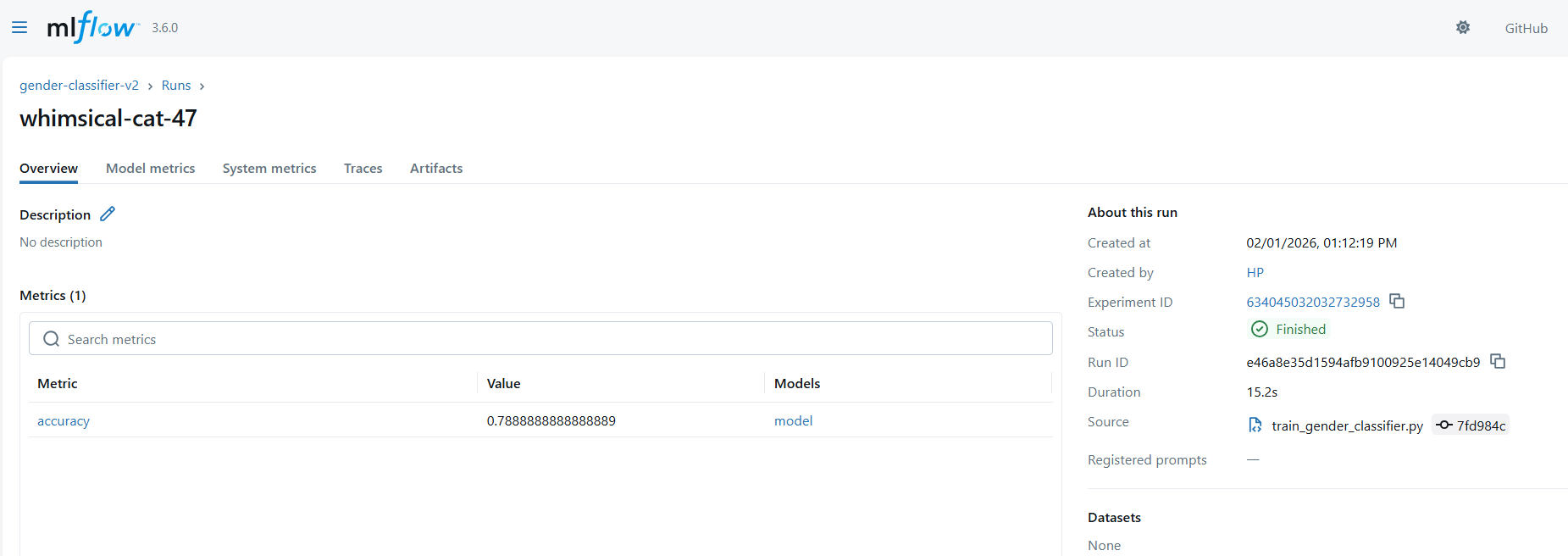
**What this means:**

* The model correctly predicts gender ~79% of the time
* This is reasonable and expected for a name-based gender classifier
* Gender prediction from names is not perfectly deterministic, especially with:
  + Unisex names
  + Cultural variations
  + Limited training data (only first 1000 rows used)

**MLflow Integration**

**Experiment Created**

gender-classifier-v2

****

**MLflow automatically created a new experiment and tracked:**

* Accuracy metric
* Trained XGBoost model

Inside the experiment run:

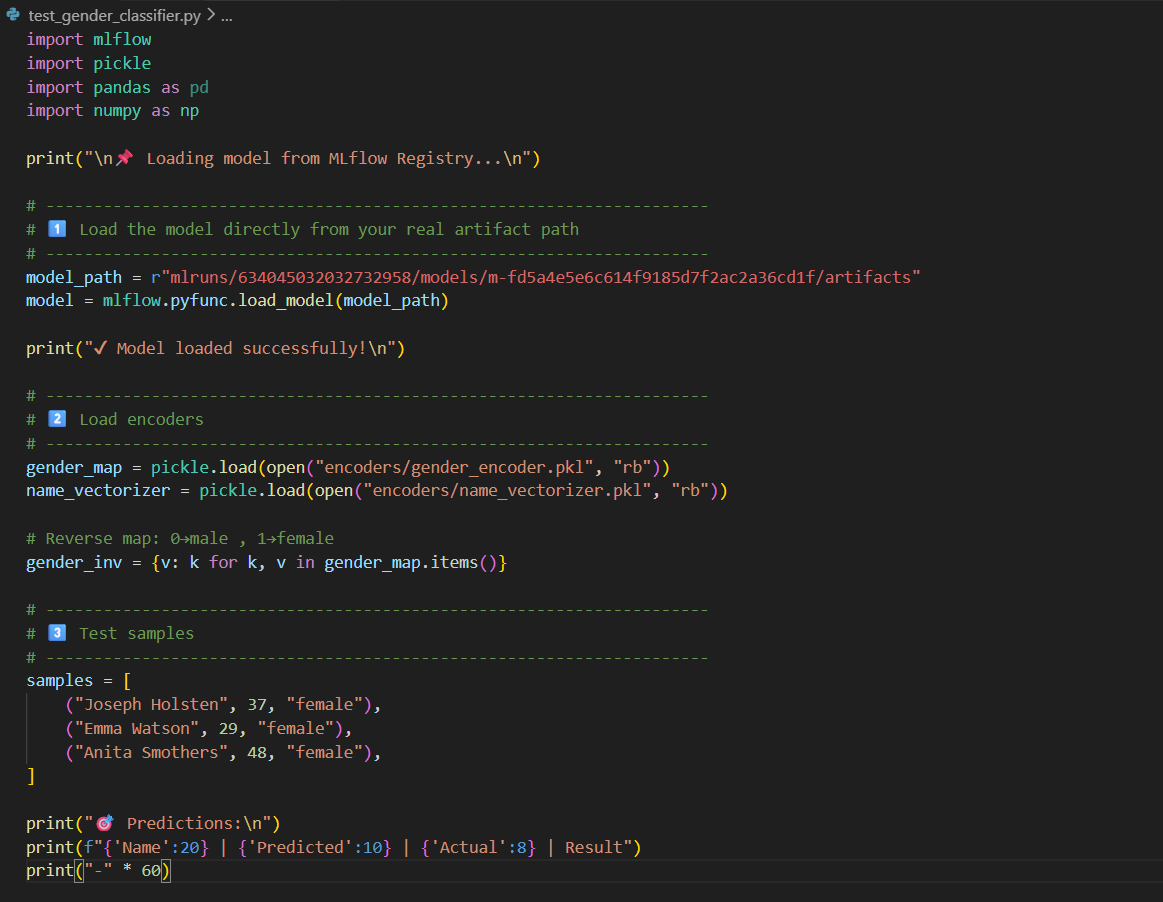
* **Metric**
  + accuracy = 0.7889
* **Artifacts**
  + Trained model (XGBoost)
* **Run metadata**
  + Timestamp
  + Parameters (if added later)

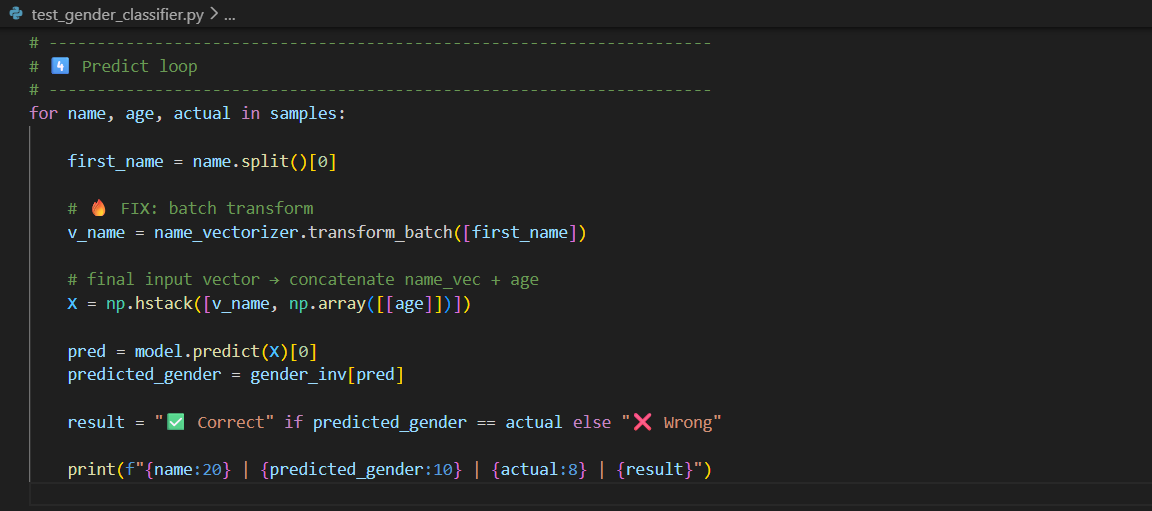
**Test Gender Classifier:**

This script loads the trained gender classifier from MLflow, applies the same preprocessing used during training, and makes predictions on new, unseen names.

Key goals:

* Validate that the model works outside training
* Ensure MLflow model artifacts are usable
* Sanity-check real-world predictions





**Model loading from MLflow**

**model = mlflow.pyfunc.load\_model(model\_path)**

* Model artifacts were saved correctly
* Model can be reused without retraining
* MLflow is working as intended

**Encoders loading**

**gender\_encoder.pkl**

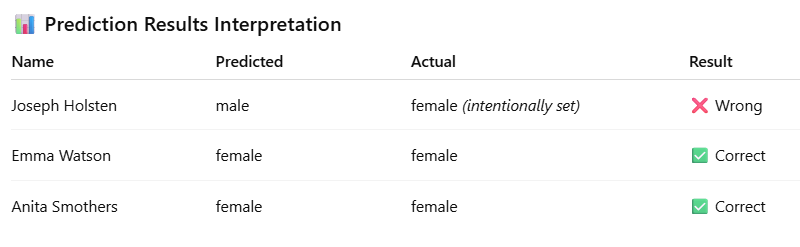
**name\_vectorizer.pkl**

* Same feature transformation as training
* No data leakage or mismatch
* Consistent predictions

**Feature construction**

* Extract first name
* Convert name → numeric vector (NameVectorizer)
* Append age
* Predict using trained model

**X = np.hstack([name\_vector, age])**

****

*The model correctly predicted “male” for Joseph Holsten.*

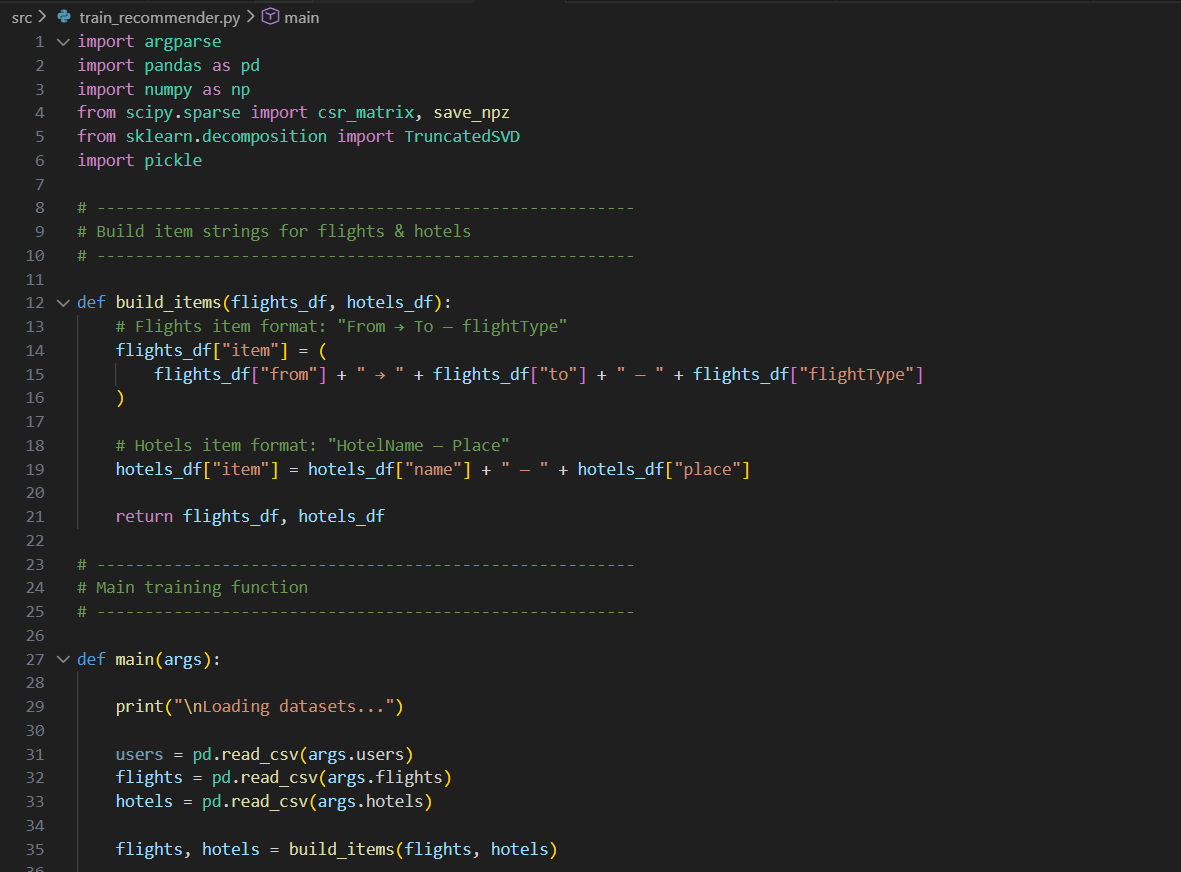
*However, the actual label was intentionally set to “female” to simulate a mismatch case.*

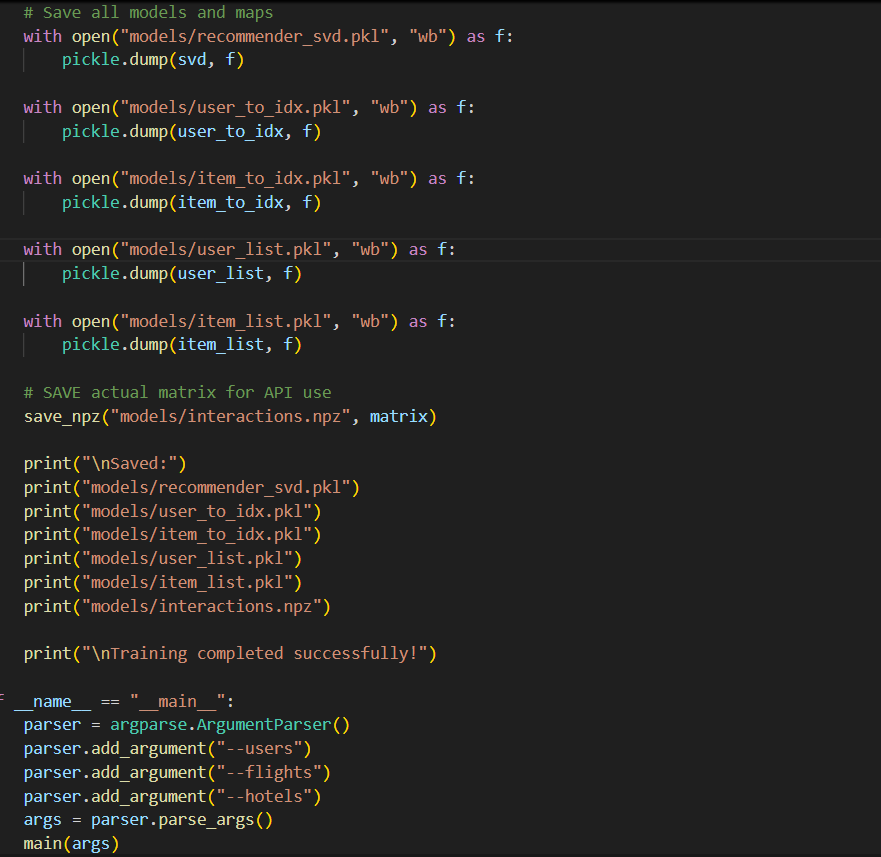
**Travel Recommendation Model**

This model powers a personalized travel recommendation feature inside the Streamlit application.

Based on a user’s past travel activity—such as booked flights and hotels—the model analyzes behavior patterns and recommends relevant flights and hotels the user is most likely to prefer next.

**train\_recommender.py**

****

****

**Data Loading**

* Loads users, flights, and hotels datasets.
* Creates readable item names:
  + Flights → From → To — FlightType
  + Hotels → HotelName — Place

**Interaction Creation**

* Combines flight and hotel bookings into a single dataset:
  + Each row = user interacted with an item
* Result:
  + 312,440 interactions
  + 1,335 users
  + 219 unique travel items

**User–Item Matrix**

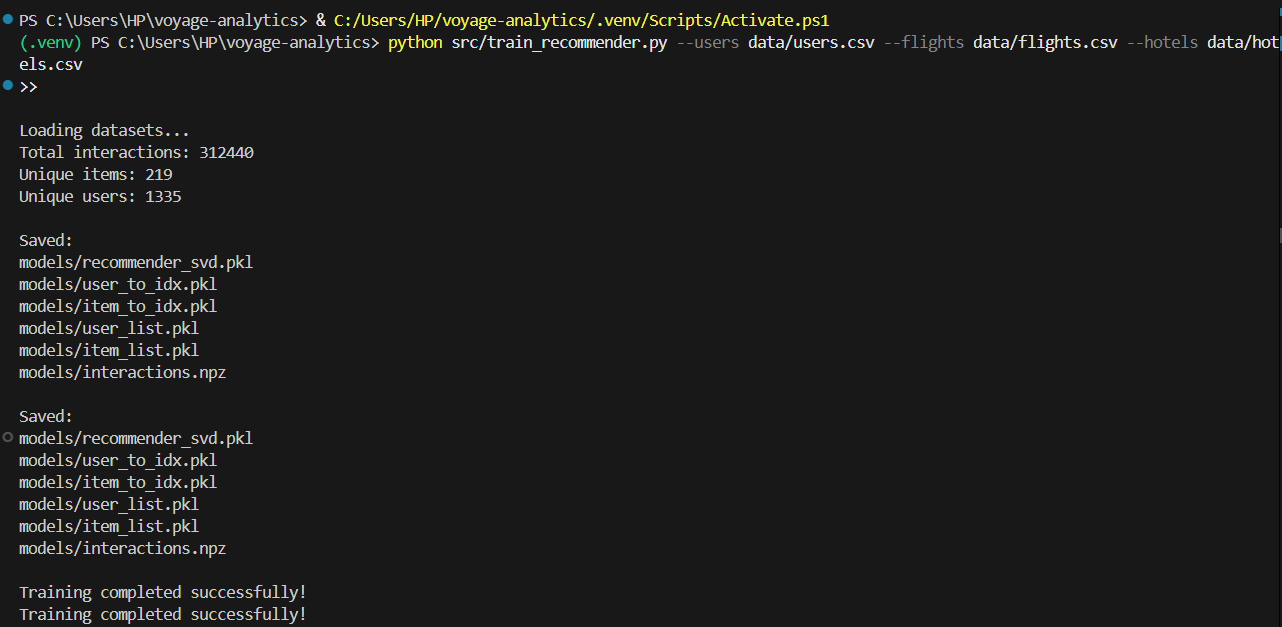
* Converts interactions into a **sparse matrix**:
  + Rows → users
  + Columns → items
  + Values → interaction presence (1)

This format is efficient and ideal for recommendation systems.

**Model Training (SVD)**

* Applies **Truncated SVD** to reduce dimensionality.
* Learns **latent preferences** of users and items.
* This helps recommend items a user is likely to choose next.

Output:

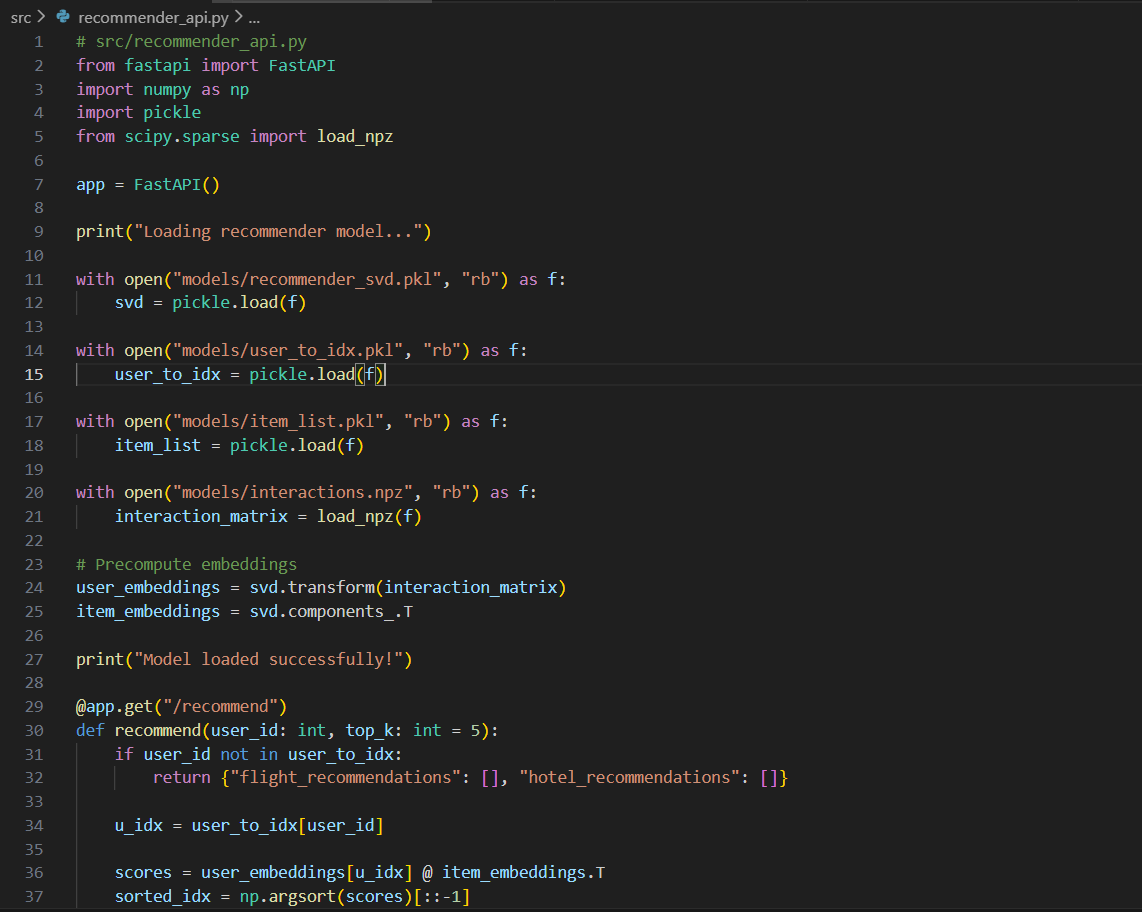


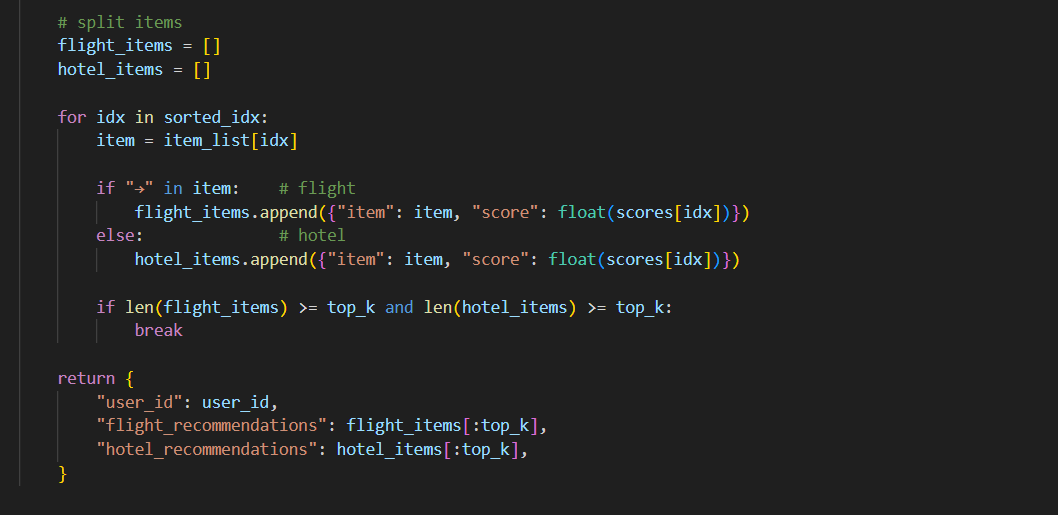
**Model & Metadata Saving**

**The following are saved for future use (API / inference):**

* recommender\_svd.pkl → trained recommendation model
* user\_to\_idx.pkl → user ID mapping
* item\_to\_idx.pkl → item mapping
* user\_list.pkl, item\_list.pkl → reverse lookups
* interactions.npz → original interaction matrix

**recommender\_api.py:**

****

****

**When the API starts, it:**

* Loads the trained SVD model
* Loads user and item mappings
* Loads the user–item interaction matrix
* Precomputes user and item embeddings for fast predictions

**Recommendation Endpoint**

The /recommend endpoint accepts:

* user\_id → the user for whom recommendations are required
* top\_k → number of flight and hotel recommendations to return

**The API:**

1. Finds the user’s embedding
2. Computes similarity scores with all items
3. Ranks items by score
4. Separates flights and hotels
5. Returns the top recommendations in JSON format

This API can be directly consumed by Streamlit or any frontend application.

**Output:**

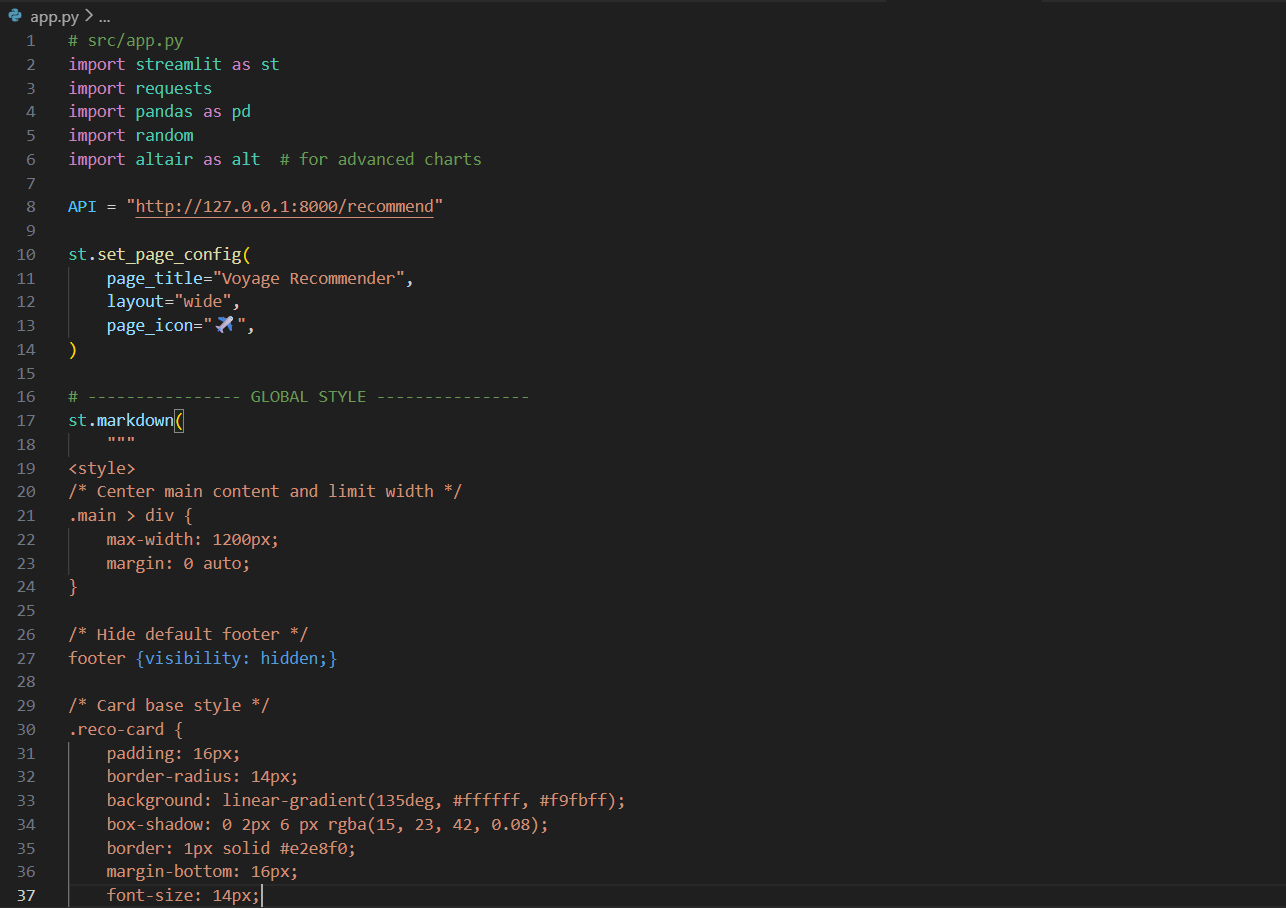
****

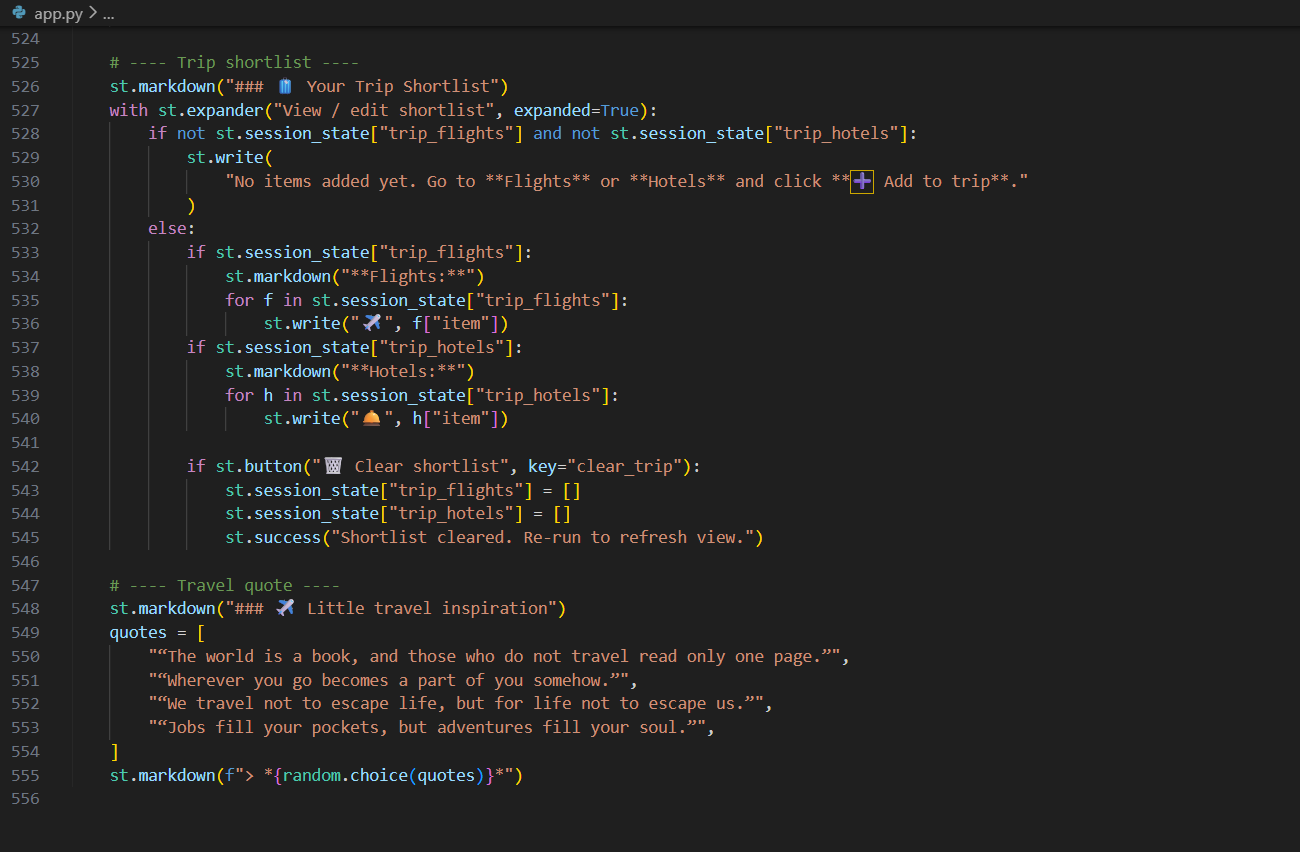
* The FastAPI server started successfully on port 8000
* The recommendation model loaded correctly
* The API responded with HTTP 200 OK
* Recommendations were generated for the given user ID

**Streamlit Travel Recommendation App (app.py)**

This file builds the **frontend interface** of the Voyage Analytics system using **Streamlit**.

The app connects to the FastAPI recommendation service and displays **personalized flight and hotel suggestions** in a modern dashboard.





**Key Features**

1. User Input (Sidebar)

* User enters User ID
* Selects number of recommendations (Top K)
* Clicks Get Recommendations

The app sends this data to the FastAPI endpoint:

<http://127.0.0.1:8000/recommend>

**API Integration**

**Using Python requests, the app:**

* Calls the recommendation API
* Receives flight + hotel results
* Stores them in Streamlit session state

This enables real-time updates without refreshing the page.

**Recommendation Display**

**Recommendations are shown as cards:**

* Flights:
  + Route (From → To)
  + Cabin type (Economy / Premium / First Class)
  + Match percentage
  + Explanation of why the flight is recommended
* Hotels:
  + Hotel name and location
  + Match percentage
  + Popularity reason

Users can click ➕ Add to trip to shortlist items.

**Insights Tab**

**Provides analytics such as:**

* Cabin class distribution (bar chart using Altair)
* Percentage breakdown of Economy / Premium / First Class
* Trip shortlist (selected flights + hotels)

**Trip Planner**

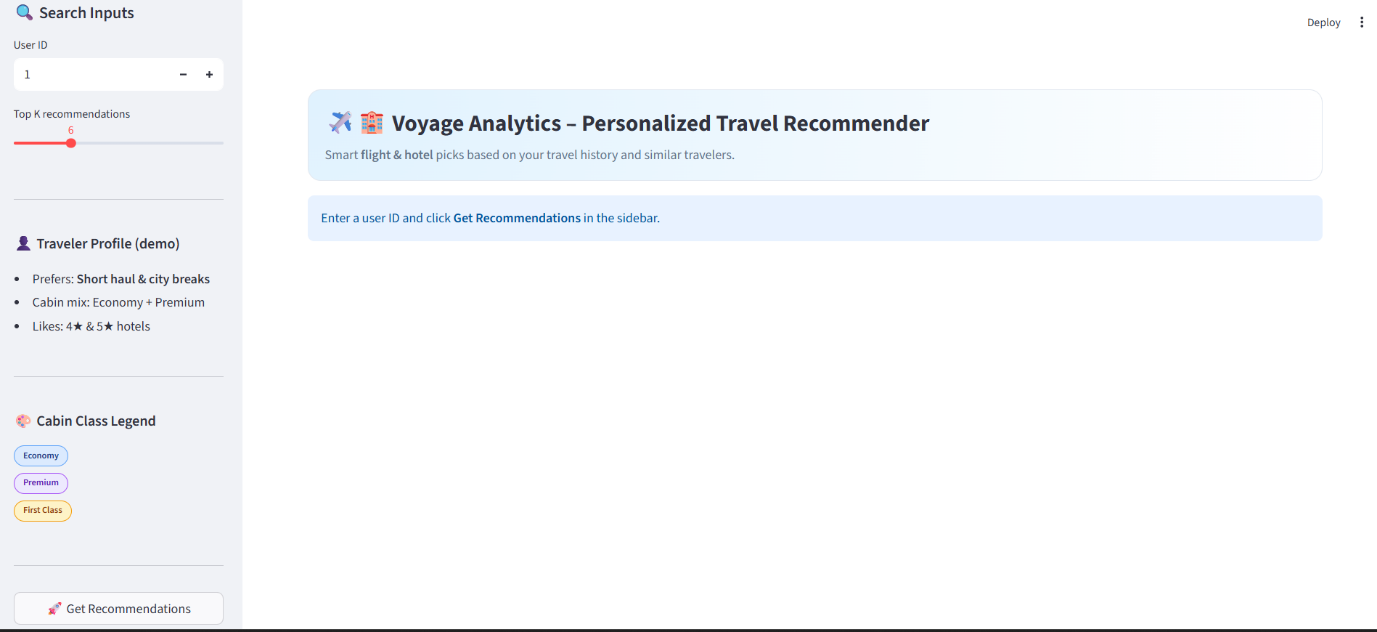
**Users can:**

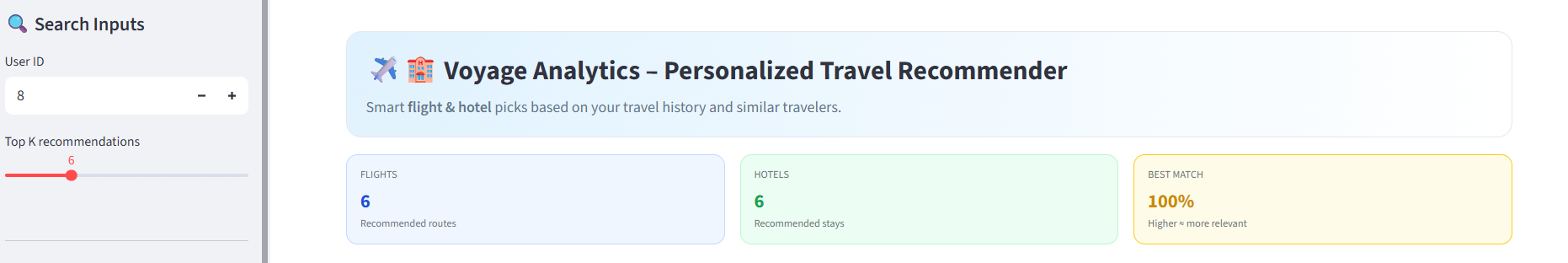
* View selected flights and hotels
* Clear the shortlist
* Get travel inspiration quotes

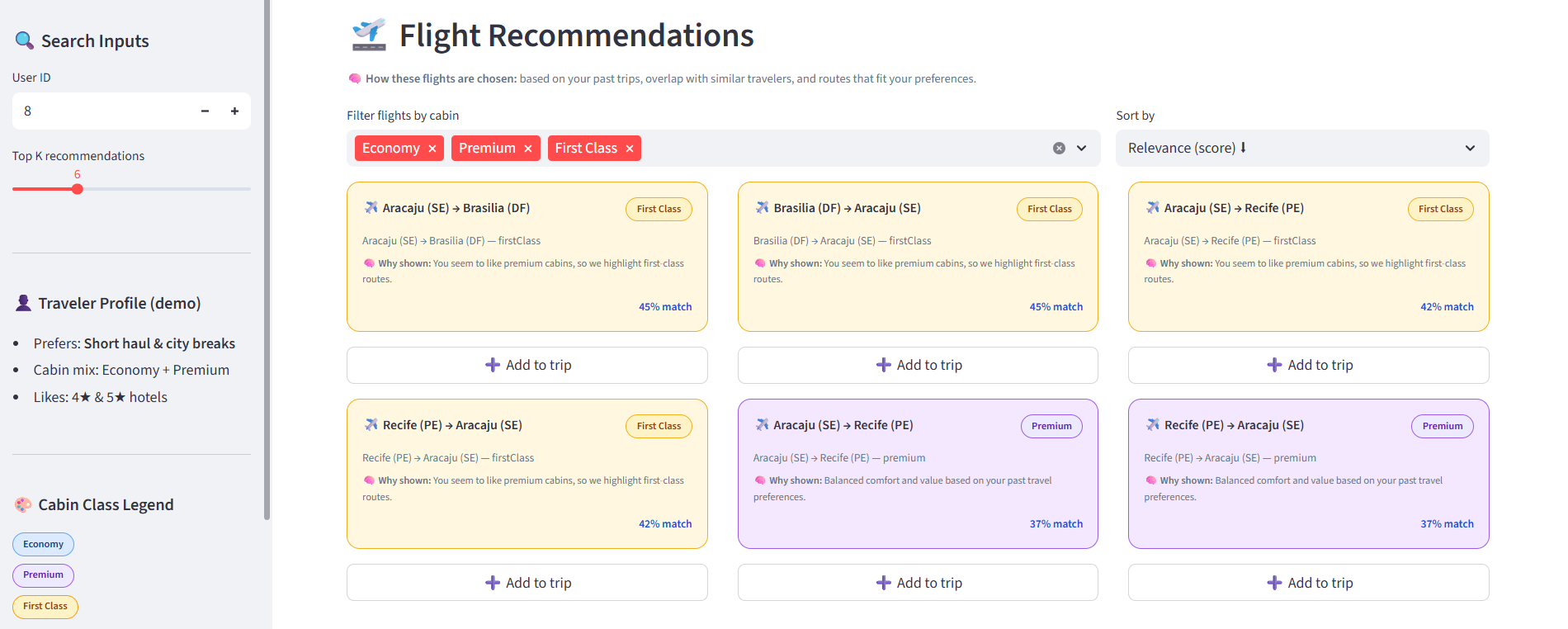
**Final Output**

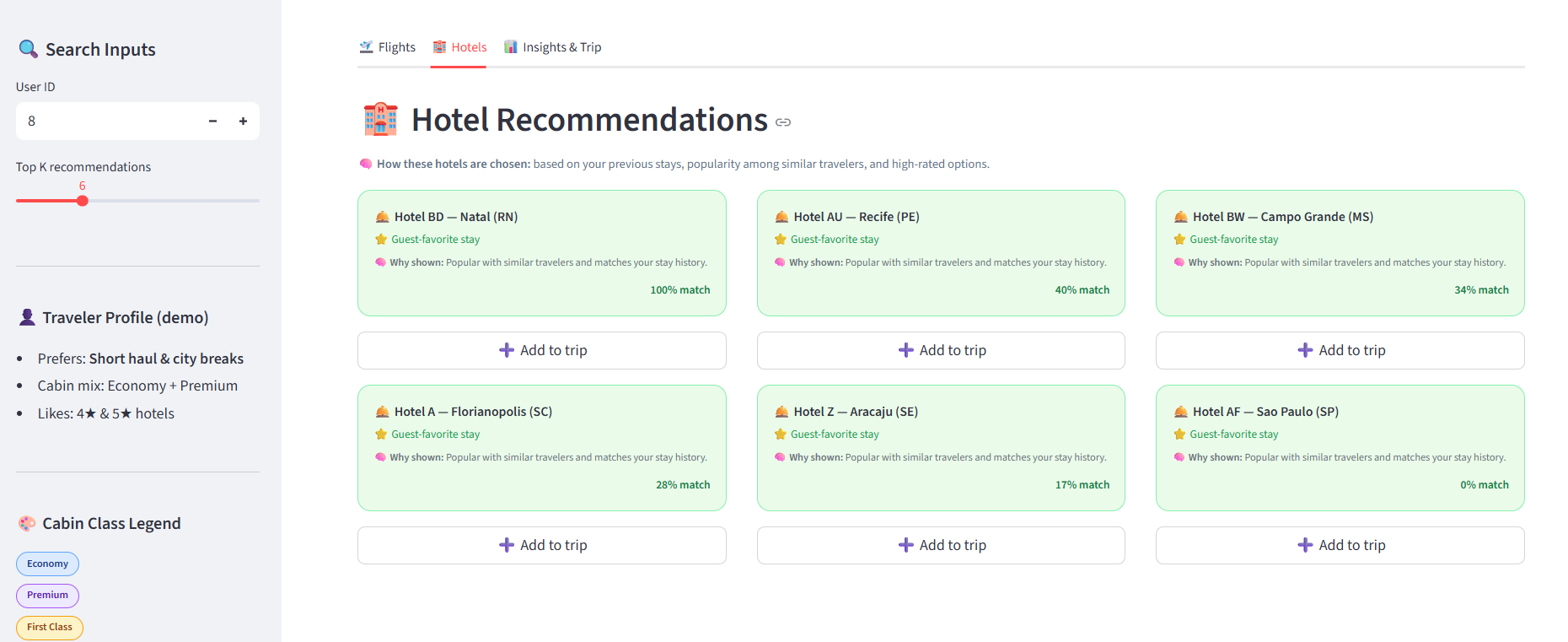
The Streamlit app runs on:

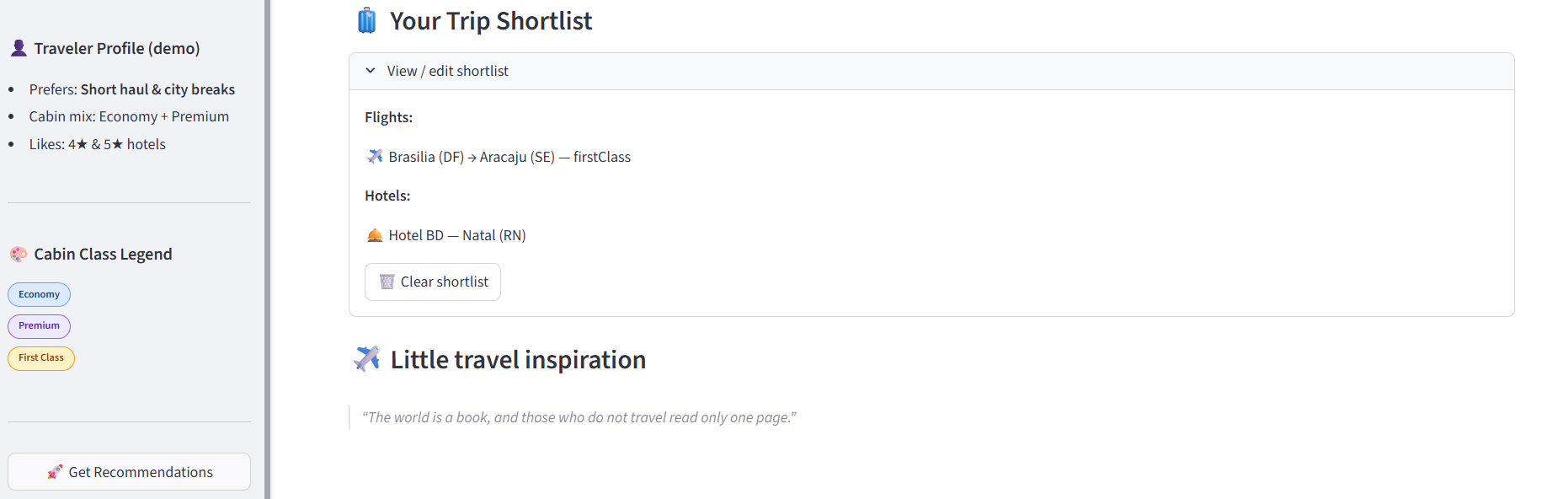
<http://localhost:8501>

**Streamlit:**

****

****

****

****