2025

EarthScape - Developer Guide

**A tree in a bubble

AI-generated content may be incorrect.**

Table of Contents

[**Student Information** 3](#_Toc208242308)

[**Management** 3](#_Toc208242309)

[**1. Developer Guide** 5](#_Toc208242311)

[**1.1 Authentication and Authorization** 5](#_Toc208242312)

[**Registering as a User (Analyst)** 5](#_Toc208242313)

[**Logging In** 7](#_Toc208242314)

[**Access Control** 10](#_Toc208242315)

[**Logging Out** 11](#_Toc208242316)

[**1.2 Data Ingestion** 12](#_Toc208242317)

[**Purpose** 12](#_Toc208242318)

[**Supported Dataset Types** 12](#_Toc208242319)

[**Steps for Data Ingestion** 12](#_Toc208242320)

[**Dataset Management** 34](#_Toc208242321)

[**1.3 Data Storage** 42](#_Toc208242322)

[**Implementation** 42](#_Toc208242323)

[**Optimization Strategies** 43](#_Toc208242324)

[**1.4 Data Processing** 45](#_Toc208242325)

[**1.5 Real-time Data Processing** 58](#_Toc208242326)

[**Steps** 58](#_Toc208242327)

[**Access Control** 70](#_Toc208242328)

[**Error Handling** 70](#_Toc208242329)

[**1.6 Anomaly Training** 71](#_Toc208242330)

[**Steps** 71](#_Toc208242331)

[**1.7 Forecast Training** 94](#_Toc208242332)

[**Steps** 94](#_Toc208242333)

[**Access Control** 97](#_Toc208242334)

[**Error Handling** 97](#_Toc208242335)

[**1.8 Correlation Analysis** 98](#_Toc208242336)

[**Features** 98](#_Toc208242337)

[**Access Control** 101](#_Toc208242338)

[**Error Handling** 101](#_Toc208242339)

[**1.10 Weather Details** 101](#_Toc208242340)

[**Features** 101](#_Toc208242341)

[**Access Control** 103](#_Toc208242342)

[**Error Handling** 103](#_Toc208242343)

[**1.11 Weather ChatBot** 103](#_Toc208242344)

[**Features** 104](#_Toc208242345)

[**Access Control** 109](#_Toc208242346)

[**1.12 Admin Management** 109](#_Toc208242347)

[**Features** 109](#_Toc208242348)

[**Access Control** 116](#_Toc208242349)

[**Error Handling** 116](#_Toc208242350)

[**1.13 Feedback Management** 116](#_Toc208242351)

[**Features** 117](#_Toc208242352)

[**Access Control** 123](#_Toc208242353)

[**Error Handling** 124](#_Toc208242354)

# **Student Information**

|  |  |
| --- | --- |
| Student ID | Student Name |
| Student1420981 | **Muhammad Saad Karim** |
| Student1421220 | **Tayyaba** |
| Student1368985 | **Mahnoor** |
| Student1418404 | **Laraib** |
| Student1335767 | **Lamia Raza** |
| Student1418410 | **Albina** |

# **Management**

|  |  |
| --- | --- |
| Role | Student Name |
| Faculty | **Mam Wajiha** |
| Coordinator | **Mam Sana** |

**ERD**

**A diagram of a diagram

AI-generated content may be incorrect.**

**System Flow**

**A diagram of a network

AI-generated content may be incorrect.**

**Real-Time SSE Sequence**

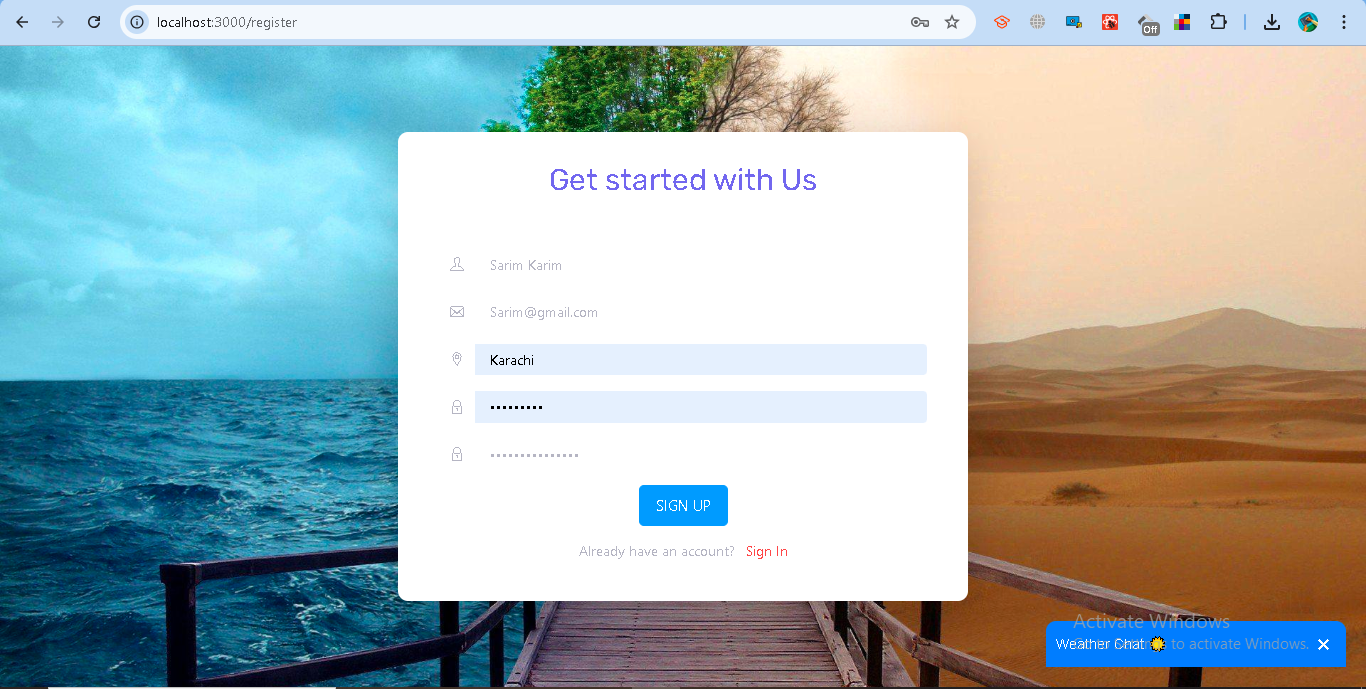
# **A diagram of a computer AI-generated content may be incorrect.**

# **1. Developer Guide**

## **1.1 Authentication and Authorization**

The system uses a secure authentication process with role-based access control (RBAC). It ensures that administrators have full rights, while analysts (normal users) have restricted access.

### **Registering as a User (Analyst)**

  
Code:

from flask import Blueprint, request, jsonify

from db import users\_collection

from passlib.hash import bcrypt

from datetime import datetime

import re

register\_bp = Blueprint("register\_bp", \_\_name\_\_)

@register\_bp.route("/register", methods=["POST"])

def register\_user():

    data = request.get\_json()

    full\_name = data.get("full\_name", "").strip()

    email = data.get("email", "").strip().lower()

    password = data.get("password", "").strip()

    city = data.get("city", "").strip()

    if not full\_name or not email or not password or not city:

        return jsonify({"error": "All fields are required"}), 400

    if not re.match(r"[^@]+@[^@]+\.[^@]+", email):

        return jsonify({"error": "Invalid email format"}), 400

    if len(password) < 6:

        return jsonify({"error": "Password must be at least 6 characters"}), 400

    if users\_collection.find\_one({"email": email}):

        return jsonify({"error": "Email already registered"}), 400

    hashed\_pw = bcrypt.hash(password)

    users\_collection.insert\_one({

        "full\_name": full\_name,

        "email": email,

        "password": hashed\_pw,

        "role": "analyst",

         "city": city,

        "created\_at": datetime.utcnow()

    })

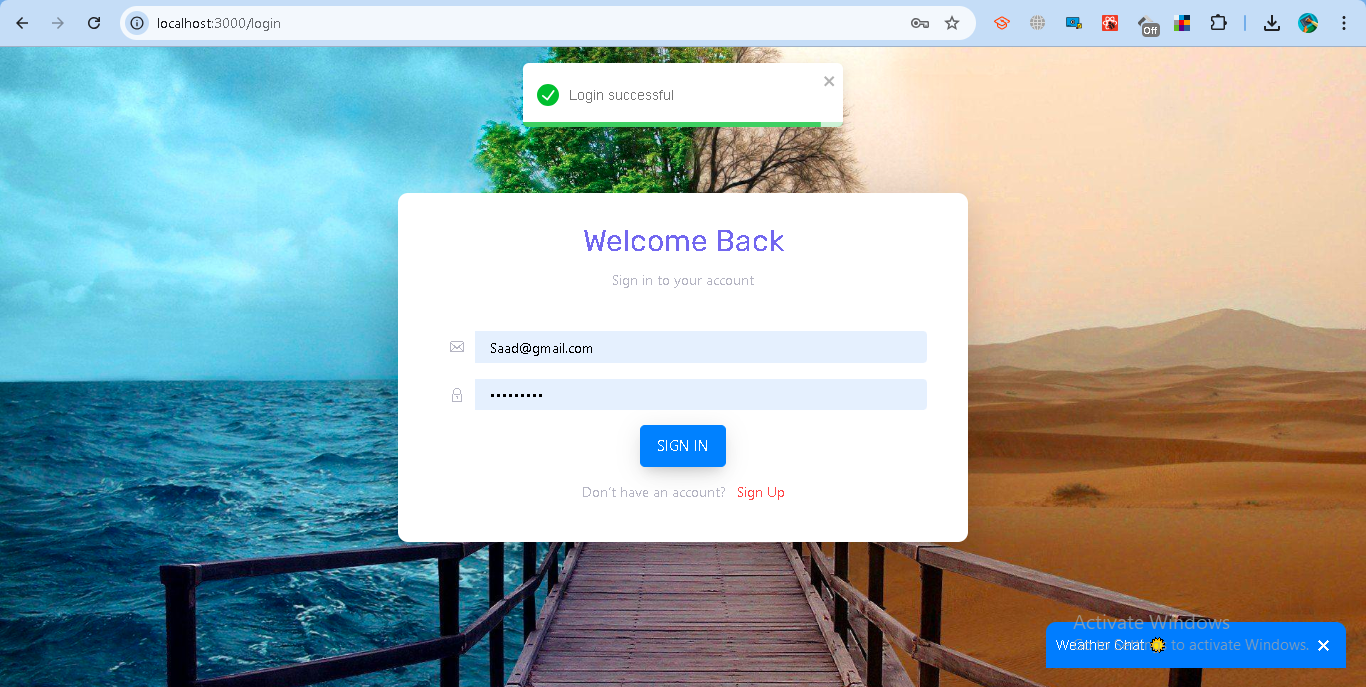
    return jsonify({"message": "Analyst registered successfully"}), 201

* Go to the registration page.
* Provide:  
  + Full Name
  + Email Address (must be valid format, e.g., user@example.com)
  + Password (minimum 6 characters, stored securely with bcrypt hashing)
  + City (for weather API integration)
* After successful registration:  
  + Your account is created with the role analyst.

**A confirmation message is shown:**  
  
 Analyst registered successfully

By default, new users are not administrators. Only system-defined admins have elevated privileges

### **Logging In**



Code:

from flask import Blueprint, request, jsonify

from db import users\_collection

from passlib.hash import bcrypt

import jwt

import os

from datetime import datetime, timedelta

login\_bp = Blueprint("login\_bp", \_\_name\_\_)

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGORITHM = os.getenv("JWT\_ALGORITHM", "HS256")

@login\_bp.route("/login", methods=["POST"])

def login():

    data = request.get\_json()

    email = data.get("email", "").strip().lower()

    password = data.get("password", "").strip()

    if not email or not password:

        return jsonify({"error": "Email and password are required"}), 400

    user = users\_collection.find\_one({"email": email})

    if not user or not bcrypt.verify(password, user["password"]):

        return jsonify({"error": "Invalid credentials"}), 401

    payload = {

         "user\_id": str(user["\_id"]),

        "email": user["email"],

        "role": user.get("role", "analyst"),

        "exp": datetime.utcnow() + timedelta(hours=2)

    }

    token = jwt.encode(payload, JWT\_SECRET, algorithm=JWT\_ALGORITHM)

    redirect\_url = "/" if user["role"] != "admin" else "/add-admin"

    return jsonify({

        "access\_token": token,

        "token\_type": "bearer",

        "role": user["role"],

           "email": user["email"],  # optional, for frontend usage

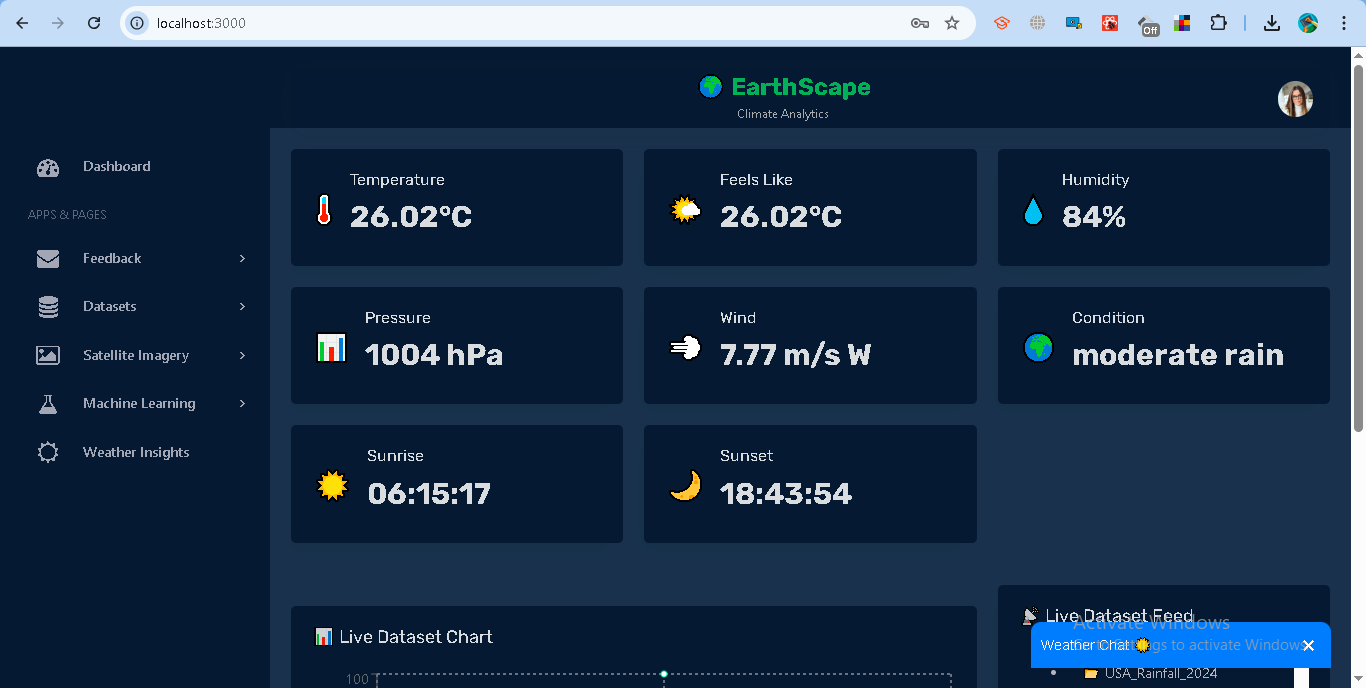
        "user\_id": str(user["\_id"]),  # convert ObjectId to string

        "redirect": redirect\_url,

        "message": "Login successful"

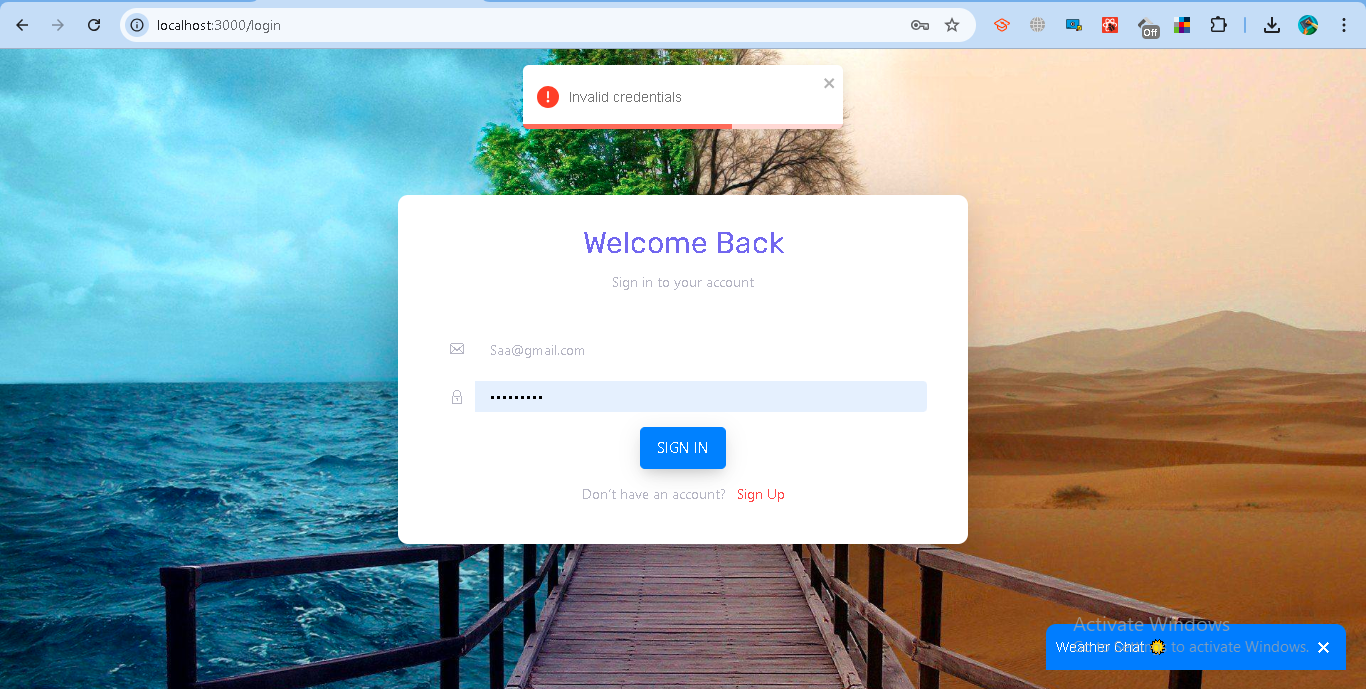
    })

* Go to the login page.
* Enter your registered email and password.
* If valid:  
  + The server generates a JWT token with:  
    - user\_id (your unique ID)
    - email
    - role (admin or analyst)
    - exp (expiry: 2 hours)
  + You will be redirected:  
    - Analyst → Dashboard (/)
    - Admin → Admin Panel (/)



* If invalid:

**Error message is shown:**  
  
 Invalid credentials

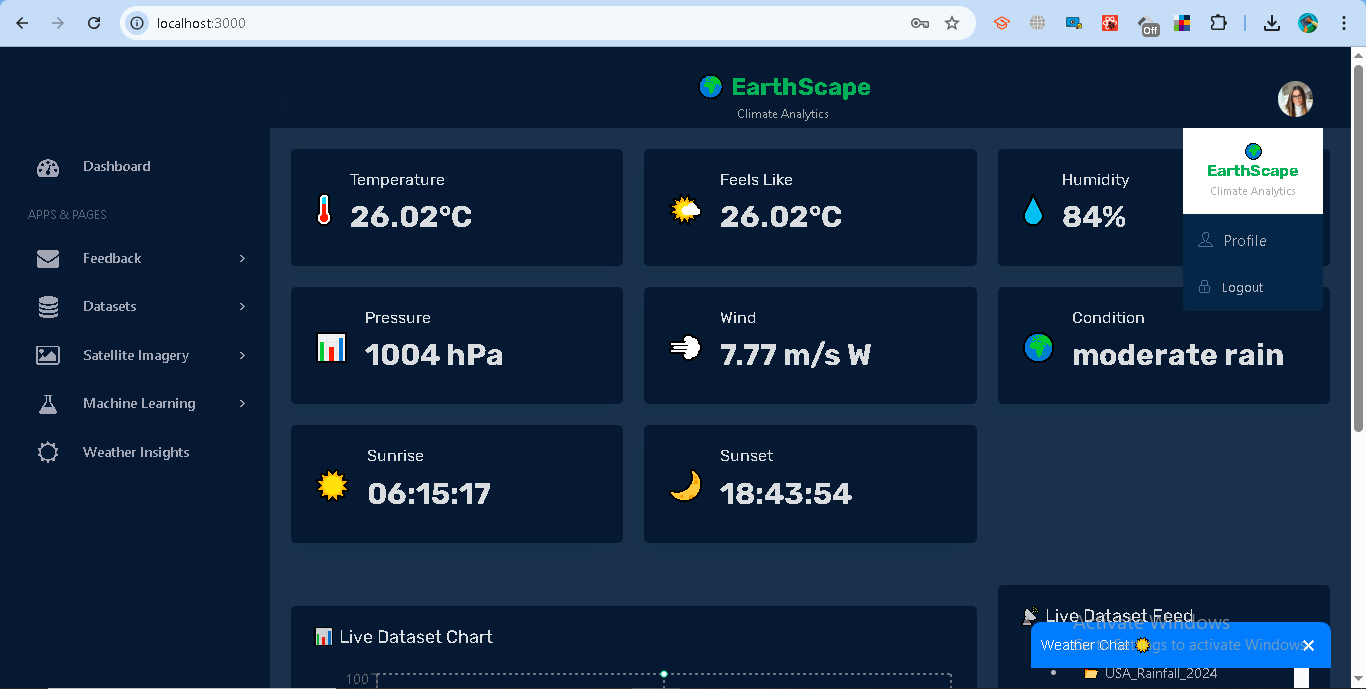


### **Access Control**

* Analyst (Normal User)  
  + Can log in and view their profile.
  + Can fetch weather details based on their city.
  + Can update their own profile information (name, email, password, city).
  + Cannot manage users or datasets.
* Administrator (Admin)  
  + Has all rights, including:  
    - Managing users (add/remove/update).
    - Accessing the Admin Panel.
    - Adding datasets.
    - Configuring alerts.
  + Can also perform all analyst actions.

### **Logging Out**

* When logging out:  
  + Your JWT token is blacklisted and cannot be reused.



Code:

from flask import Blueprint, jsonify, request

logout\_bp = Blueprint("logout", \_\_name\_\_)

# In-memory blacklist

blacklisted\_tokens = set()

@logout\_bp.route("/logout", methods=["POST"])

def logout():

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Authorization header missing"}), 401

    token = auth\_header.split(" ")[1]

    # Add token to blacklist immediately

    blacklisted\_tokens.add(token)

    return jsonify({"message": "Logged out successfully"}), 200

def is\_token\_blacklisted(token):

    return token in blacklisted\_tokens

**Response from backend:**  
  
 Logged out successfully

This ensures secure login, session management, and role-based permissions. Users cannot escalate privileges since role is assigned at registration and enforced in the backend.

# **1.2 Data Ingestion**

### **Purpose**

The data ingestion module allows users to upload and manage climate-related datasets such as CSV files, satellite imagery, and weather records. This ensures that both historical and real-time data sources can be stored, processed, and later analyzed.

### **Supported Dataset Types**

1. **CSV Upload** Users can upload their own structured datasets in .csv format.
2. **Satellite Imagery** Users can fetch and upload remote sensing imagery based on location (latitude/longitude) and zoom level.
3. **Weather Data (Historical)** Users can pull climate records (temperature, humidity, wind speed, precipitation, etc.) from the Meteostat API by specifying a city and date range.

### **Steps for Data Ingestion**

#### **A. Uploading a CSV Dataset**

1. Login to the system using your credentials.
2. Go to **Dataset → Upload Dataset → CSV.**
3. Enter a dataset name (example: USA\_Rainfall\_2024).
4. Select and upload a .csv file containing your climate data.



Code: from flask import Blueprint, request, jsonify, send\_from\_directory

from db import db

import jwt

import os

from bson import ObjectId

datasets\_bp = Blueprint("datasets\_bp", \_\_name\_\_)

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGORITHM = os.getenv("JWT\_ALGORITHM", "HS256")

DATASET\_FOLDER = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), "dataset")  # folder where CSVs are stored

BASE\_URL = os.getenv("BASE\_URL", "http://127.0.0.1:5000")   # backend URL

# 📌 Get datasets

@datasets\_bp.route("/datasets", methods=["GET"])

def get\_datasets():

    # 1️⃣ Check Authorization header

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Authorization header missing"}), 401

    try:

        token = auth\_header.split(" ")[1]

        decoded = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        user\_id = decoded.get("user\_id")

        role = decoded.get("role")

        if not user\_id or not role:

            return jsonify({"error": "Invalid token payload"}), 401

    except jwt.ExpiredSignatureError:

        return jsonify({"error": "Token expired"}), 401

    except jwt.InvalidTokenError:

        return jsonify({"error": "Invalid token"}), 401

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    try:

        datasets\_collection = db.datasets

        # 2️⃣ Fetch datasets based on role

        if role == "admin":

            users\_datasets = list(datasets\_collection.find({}))

        else:

            users\_datasets = list(datasets\_collection.find({"user\_id": ObjectId(user\_id)}))

        # 3️⃣ Format response

        response\_data = []

        for user\_doc in users\_datasets:

            user\_dict = {

                "\_id": str(user\_doc["\_id"]),

                "user\_id": str(user\_doc.get("user\_id", "")),

                "datasets": []

            }

            for ds in user\_doc.get("datasets", []):

                filename = ds.get("filename")

                download\_url = (

                    f"{BASE\_URL}/api/datasets/download/{filename}"

                    if filename else None

                )

                user\_dict["datasets"].append({

                    "dataset\_name": ds.get("dataset\_name", "N/A"),

                    "dataset\_type": ds.get("dataset\_type", "N/A"),

                    "filename": filename or "N/A",

                    "file\_path": ds.get("file\_path", "N/A"),

                    "uploaded\_at": str(ds.get("uploaded\_at", "N/A")),

                    "records\_count": len(ds.get("records", [])) if ds.get("records") else 0,

                    "uploaded\_by": ds.get("uploaded\_by", "N/A"),

                    "download\_url": download\_url

                })

            response\_data.append(user\_dict)

        return jsonify({"datasets": response\_data}), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

# 📌 Get imagery datasets only

@datasets\_bp.route("/datasets/imagery", methods=["GET"])

def get\_imagery\_datasets():

    # 1️⃣ Check Authorization header

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Authorization header missing"}), 401

    try:

        token = auth\_header.split(" ")[1]

        decoded = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        user\_id = decoded.get("user\_id")

        role = decoded.get("role")

        if not user\_id or not role:

            return jsonify({"error": "Invalid token payload"}), 401

    except jwt.ExpiredSignatureError:

        return jsonify({"error": "Token expired"}), 401

    except jwt.InvalidTokenError:

        return jsonify({"error": "Invalid token"}), 401

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    try:

        satellite\_imagery\_collection = db.satellite\_imagery

        # 2️⃣ Fetch documents

        if role == "admin":

            imagery\_docs = list(satellite\_imagery\_collection.find({}))

        else:

            imagery\_docs = list(satellite\_imagery\_collection.find({"user\_id": ObjectId(user\_id)}))

        # 3️⃣ Format response

        response\_data = []

        for doc in imagery\_docs:

            user\_id = str(doc.get("user\_id", ""))

            datasets = doc.get("satellite\_imagery", [])

            for dataset in datasets:  # loop inside nested array

                filename = dataset.get("filename")

                download\_url = (

                    f"{BASE\_URL}/api/datasets/download/{filename}"

                    if filename else None

                )

                response\_data.append({

                    "\_id": str(doc["\_id"]),

                    "user\_id": user\_id,

                    "dataset\_name": dataset.get("dataset\_name", "N/A"),

                    "dataset\_type": dataset.get("dataset\_type", "N/A"),

                    "filename": filename or "N/A",

                    "file\_path": dataset.get("file\_path", "N/A"),

                    "uploaded\_at": str(dataset.get("uploaded\_at", "N/A")),

                    "uploaded\_by": dataset.get("uploaded\_by", "N/A"),

                    "records\_count": dataset.get("records\_count", 0)

                        if "records\_count" in dataset else len(dataset.get("records", [])),

                    "gibs\_url": dataset.get("gibs\_url"),

                    "download\_url": download\_url,

                    # ✅ Include the full records array

                    "records": dataset.get("records", [])

                })

        return jsonify({"datasets": response\_data}), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

# 🗑️ Delete imagery dataset route

@datasets\_bp.route("/datasets/imagery/<doc\_id>/<filename>", methods=["DELETE"])

def delete\_imagery\_dataset(doc\_id, filename):

    # 1️⃣ Check Authorization header

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Authorization header missing"}), 401

    try:

        token = auth\_header.split(" ")[1]

        decoded = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        user\_id = decoded.get("user\_id")

        role = decoded.get("role")

        if not user\_id or not role:

            return jsonify({"error": "Invalid token payload"}), 401

    except jwt.ExpiredSignatureError:

        return jsonify({"error": "Token expired"}), 401

    except jwt.InvalidTokenError:

        return jsonify({"error": "Invalid token"}), 401

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    try:

        imagery\_collection = db.satellite\_imagery

        # 2️⃣ Only allow user to delete their own imagery unless admin

        query = {"\_id": ObjectId(doc\_id)} if role == "admin" else {

            "\_id": ObjectId(doc\_id),

            "user\_id": ObjectId(user\_id)

        }

        imagery\_doc = imagery\_collection.find\_one(query)

        if not imagery\_doc:

            return jsonify({"error": "Imagery document not found or not authorized"}), 404

        # 3️⃣ Pull from the nested "satellite\_imagery" array

        updated = imagery\_collection.update\_one(

            query,

            {"$pull": {"satellite\_imagery": {"filename": filename}}}

        )

        if updated.modified\_count == 0:

            return jsonify({"error": "Imagery dataset not found in records"}), 404

        # 4️⃣ Delete actual file if it exists

        file\_path = os.path.join(DATASET\_FOLDER, filename)

        if os.path.exists(file\_path):

            os.remove(file\_path)

        return jsonify({"message": f"Imagery dataset '{filename}' deleted successfully"}), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

# ✅ File download route

@datasets\_bp.route("/datasets/download/<filename>", methods=["GET"])

def download\_dataset(filename):

    try:

        return send\_from\_directory(DATASET\_FOLDER, filename, as\_attachment=True)

    except FileNotFoundError:

        return jsonify({"error": "File not found"}), 404

# 🗑️ Delete dataset route

@datasets\_bp.route("/datasets/<dataset\_id>/<filename>", methods=["DELETE"])

def delete\_dataset(dataset\_id, filename):

    # 1️⃣ Check Authorization header

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Authorization header missing"}), 401

    try:

        token = auth\_header.split(" ")[1]

        decoded = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        user\_id = decoded.get("user\_id")

        role = decoded.get("role")

        if not user\_id or not role:

            return jsonify({"error": "Invalid token payload"}), 401

    except jwt.ExpiredSignatureError:

        return jsonify({"error": "Token expired"}), 401

    except jwt.InvalidTokenError:

        return jsonify({"error": "Invalid token"}), 401

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    try:

        datasets\_collection = db.datasets

        # 2️⃣ Only allow user to delete their own dataset unless admin

        query = {"\_id": ObjectId(dataset\_id)} if role == "admin" else {

            "\_id": ObjectId(dataset\_id),

            "user\_id": ObjectId(user\_id)

        }

        dataset\_doc = datasets\_collection.find\_one(query)

        if not dataset\_doc:

            return jsonify({"error": "Dataset not found or not authorized"}), 404

        # 3️⃣ Remove dataset entry from MongoDB

        updated = datasets\_collection.update\_one(

            query,

            {"$pull": {"datasets": {"filename": filename}}}

        )

        if updated.modified\_count == 0:

            return jsonify({"error": "Dataset not found in user records"}), 404

        # 4️⃣ Delete actual file if it exists

        file\_path = os.path.join(DATASET\_FOLDER, filename)

        if os.path.exists(file\_path):

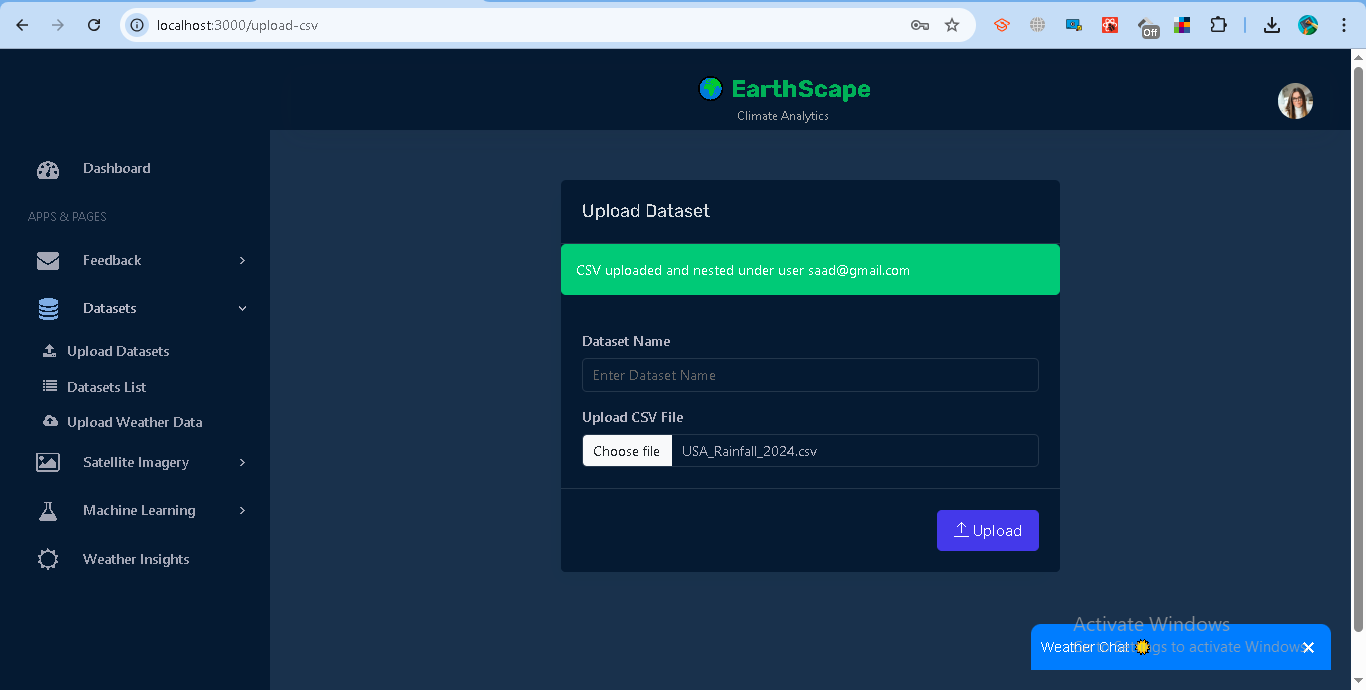
            os.remove(file\_path)

        return jsonify({"message": f"Dataset '{filename}' deleted successfully"}), 200

    except Exception as e:

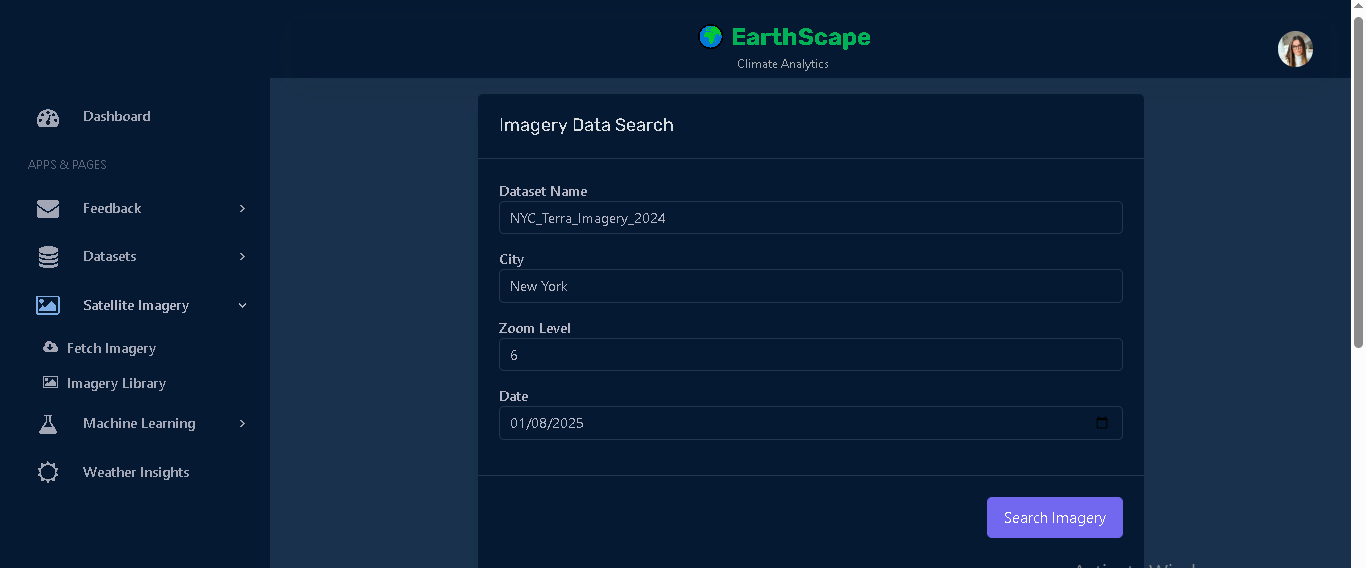
        return jsonify({"error": str(e)}), 500

1. The system will:  
   * Validate the file.
   * Save records in the database.
   * Provide feedback on the number of records uploaded.
2. A confirmation message will be displayed:  
     
     
    *“CSV uploaded and nested under user [your email].”*

**

#### **B. Uploading Satellite Imagery**

1. Go to **Satellite Imagery→ Fetch Imagery.**
2. Provide:  
   * Dataset name.
   * City name.
   * Latitude and Longitude coordinates.
   * Zoom level (higher = more detail).
   * Date (or leave blank to use today).



Code:

import os

import jwt

import math

import pandas as pd

from flask import Blueprint, request, jsonify

from bson import ObjectId

from db import db, users\_collection

from datetime import datetime, timezone

upload\_imagery\_bp = Blueprint("upload\_imagery", \_\_name\_\_)

DATASET\_FOLDER = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), "dataset")

os.makedirs(DATASET\_FOLDER, exist\_ok=True)

def get\_user\_from\_token():

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return None, None, "Authorization header missing"

    try:

        token = auth\_header.split(" ")[1]

        decoded = jwt.decode(token, os.getenv("JWT\_SECRET"), algorithms=["HS256"])

        user\_id = decoded.get("user\_id")

        if not user\_id:

            return None, None, "user\_id not found in token"

        user = users\_collection.find\_one({"\_id": ObjectId(user\_id)})

        if not user:

            return None, None, "User not found"

        return ObjectId(user\_id), user["email"], None

    except jwt.ExpiredSignatureError:

        return None, None, "Token expired"

    except jwt.InvalidTokenError:

        return None, None, "Invalid token"

    except Exception as e:

        return None, None, str(e)

def latlon\_to\_tile(lat, lon, z):

    """Convert latitude/longitude to WMTS tile coordinates (EPSG:4326)."""

    n = 2 \*\* (z - 1)  # EPSG:4326 has half the rows compared to EPSG:3857

    x = int((lon + 180.0) / 360.0 \* (2 \*\* z))

    y = int((90.0 - lat) / 180.0 \* n)

    return x, y

@upload\_imagery\_bp.route("/upload-imagery", methods=["POST"])

def upload\_imagery():

    try:

        # ✅ Auth check

        user\_id, user\_email, error = get\_user\_from\_token()

        if error:

            return jsonify({"error": error}), 401

        data = request.json

        dataset\_name = data.get("dataset\_name")

        city = data.get("city")

        lat = data.get("latitude")

        lon = data.get("longitude")

        z = int(data.get("zoom", 6))

        date = data.get("date", datetime.now().strftime("%Y-%m-%d"))

        if not dataset\_name or not city or lat is None or lon is None:

            return jsonify({"error": "dataset\_name, city, latitude, longitude are required"}), 400

        # ✅ Convert lat/lon to x/y

        x, y = latlon\_to\_tile(float(lat), float(lon), z)

        # ✅ Construct GIBS WMTS URL

        gibs\_url = (

            f"https://gibs.earthdata.nasa.gov/wmts/epsg4326/best/"

            f"MODIS\_Terra\_CorrectedReflectance\_TrueColor/default/{date}/250m/{z}/{y}/{x}.jpg"

        )

        # ✅ Record

        records = [{

            "datetime": datetime.now(timezone.utc).isoformat(),

            "city": city,

            "latitude": float(lat),

            "longitude": float(lon),

            "zoom": z,

            "x": x,

            "y": y,

            "date": date,

            "gibs\_url": gibs\_url,

        }]

        # ✅ Save CSV file

        df = pd.DataFrame(records)

        filename = f"{city}\_imagery.csv"

        file\_path = os.path.join(DATASET\_FOLDER, filename)

        df.to\_csv(file\_path, index=False)

        # ✅ Save in MongoDB

        db.satellite\_imagery.update\_one(

            {"user\_id": user\_id},

            {

                "$push": {

                    "satellite\_imagery": {

                        "dataset\_name": dataset\_name,

                         "dataset\_type": "imagery",

                        "filename": filename,

                        "file\_path": file\_path,

                        "uploaded\_at": datetime.now(timezone.utc),

                        "uploaded\_by": user\_email,

                        "records": records,

                        "records\_count": len(records),

                    }

                }

            },

            upsert=True

        )

        return jsonify({

            "message": f"Imagery for {city} uploaded successfully.",

            "dataset\_name": dataset\_name,

            "dataset\_type": "imagery",

            "records\_count": len(records),

            "csv\_file": filename,

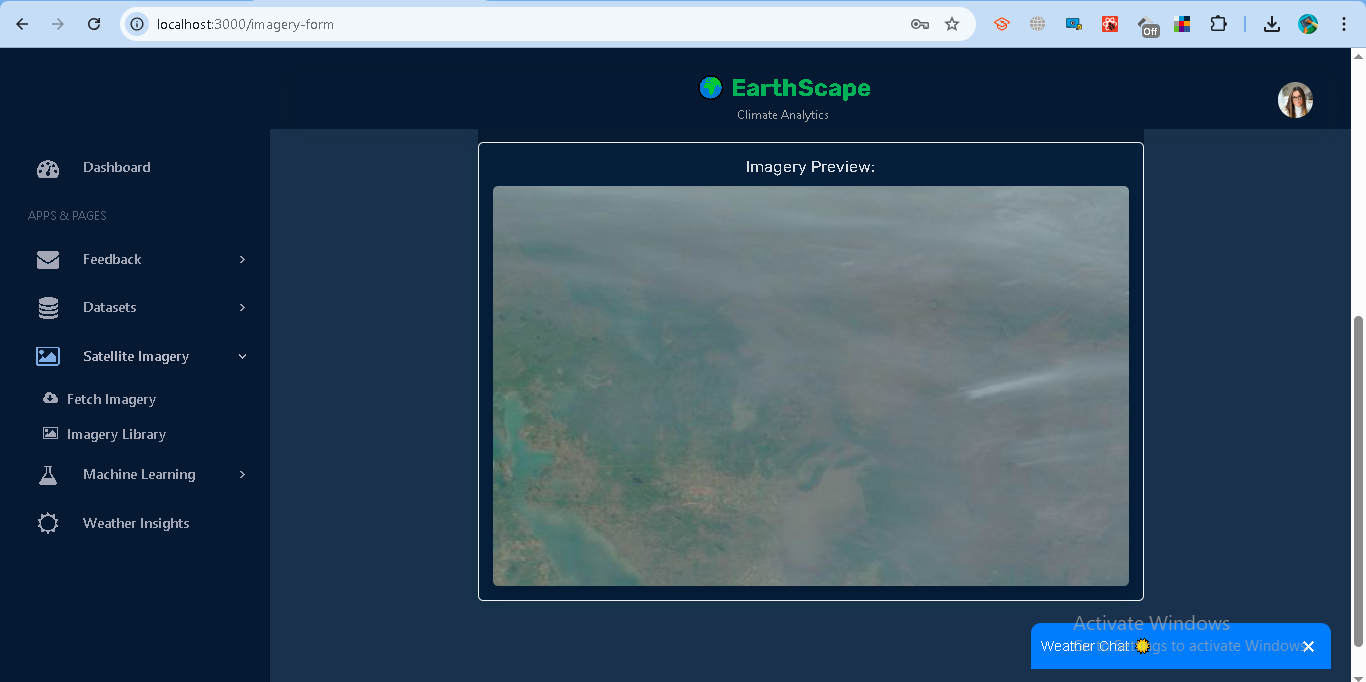
            "preview\_url": gibs\_url

        }), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

1. The system converts your location into WMTS tile coordinates and fetches an image from NASA’s GIBS API.
2. Data is stored with metadata (location, date, zoom, and GIBS preview URL).
3. A CSV file with metadata is saved, and a confirmation message appears:



#### **C. Uploading Weather Data**

1. Go to **Dataset → Uploade Weather Data**.
2. Provide:  
   * Dataset name.
   * City name.
   * Start and End date.



Code:

import os

import jwt

import math

import pandas as pd

from flask import Blueprint, request, jsonify

from bson import ObjectId

from db import db, users\_collection

from datetime import datetime, timezone

from meteostat import Point, Daily

upload\_weather\_bp = Blueprint("upload\_weather", \_\_name\_\_)

DATASET\_FOLDER = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), "dataset")

os.makedirs(DATASET\_FOLDER, exist\_ok=True)   # make sure folder exists

def clean\_value(val):

    """Ensure MongoDB-safe values (replace <NA>, NaN, etc with None)."""

    if val is None:

        return None

    if val is pd.NA:

        return None

    if isinstance(val, float) and math.isnan(val):

        return None

    return val

def get\_user\_from\_token():

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return None, None, "Authorization header missing"

    try:

        token = auth\_header.split(" ")[1]

        decoded = jwt.decode(token, os.getenv("JWT\_SECRET"), algorithms=["HS256"])

        user\_id = decoded.get("user\_id")

        if not user\_id:

            return None, None, "user\_id not found in token"

        user = users\_collection.find\_one({"\_id": ObjectId(user\_id)})

        if not user:

            return None, None, "User not found"

        return ObjectId(user\_id), user["email"], None

    except jwt.ExpiredSignatureError:

        return None, None, "Token expired"

    except jwt.InvalidTokenError:

        return None, None, "Invalid token"

    except Exception as e:

        return None, None, str(e)

@upload\_weather\_bp.route("/upload-weather", methods=["POST"])

def upload\_weather():

    try:

        user\_id, user\_email, error = get\_user\_from\_token()

        if error:

            return jsonify({"error": error}), 401

        data = request.json

        dataset\_name = data.get("dataset\_name")

        latitude = data.get("latitude")

        longitude = data.get("longitude")

        start\_date = data.get("start\_date")

        end\_date = data.get("end\_date")

        city = data.get("city")

        if not dataset\_name or not latitude or not longitude or not start\_date or not end\_date:

            return jsonify({"error": "dataset\_name, city, latitude, longitude, start\_date, end\_date are required"}), 400

        # Meteostat Point

        location = Point(float(latitude), float(longitude))

        # Fetch daily historical weather

        start = datetime.fromisoformat(start\_date)

        end = datetime.fromisoformat(end\_date)

        raw\_data = Daily(location, start, end).fetch()

        if raw\_data.empty:

            return jsonify({"error": "No historical weather data found"}), 404

        # Build safe records

        records = []

        for date, row in raw\_data.iterrows():

            record = {

                "datetime": date.strftime("%Y-%m-%d"),

                "city": city,

                "latitude": float(latitude),

                "longitude": float(longitude),

                "temperature": clean\_value(row.get("tavg")),

                "temp\_min": clean\_value(row.get("tmin")),

                "temp\_max": clean\_value(row.get("tmax")),

                "humidity": clean\_value(row.get("rhum")),

                "pressure": clean\_value(row.get("pres")),

                "wind\_speed": clean\_value(row.get("wspd")),

                "wind\_deg": clean\_value(row.get("wdir")),

                "precipitation": clean\_value(row.get("prcp")),

                "description": None

            }

            records.append(record)

        # ✅ Save CSV file

        df = pd.DataFrame(records)

        filename = f"{city}\_historical\_weather.csv"

        file\_path = os.path.join(DATASET\_FOLDER, filename)

        df.to\_csv(file\_path, index=False)

        # ✅ Save in MongoDB

        db.datasets.update\_one(

            {"user\_id": user\_id},

            {

                "$push": {

                    "datasets": {

                        "dataset\_name": dataset\_name,

                        "filename": filename,   # now CSV

                        "file\_path": file\_path,

                        "uploaded\_at": datetime.now(timezone.utc),

                        "uploaded\_by": user\_email,

                        "records": records,

                        "records\_count": len(records)

                    }

                }

            },

            upsert=True

        )

        return jsonify({

            "message": f"Historical weather data for {city} uploaded successfully.",

            "dataset\_name": dataset\_name,

            "records\_count": len(records),

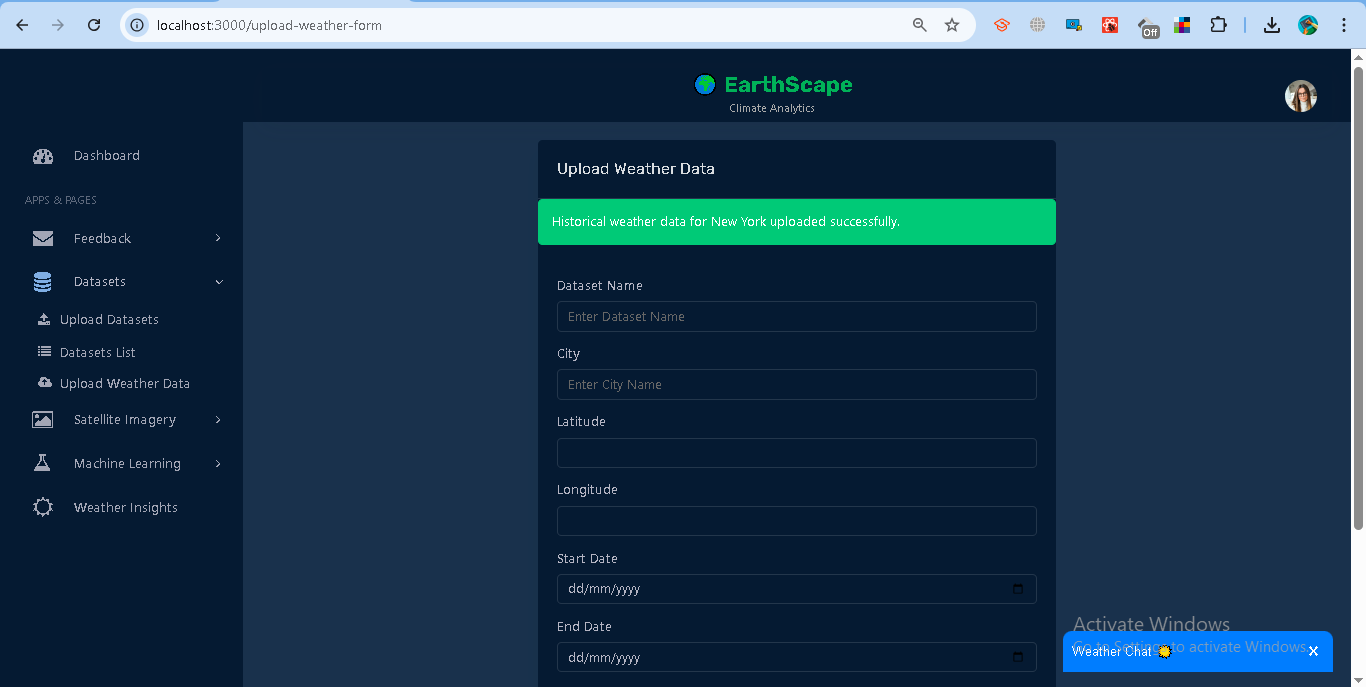
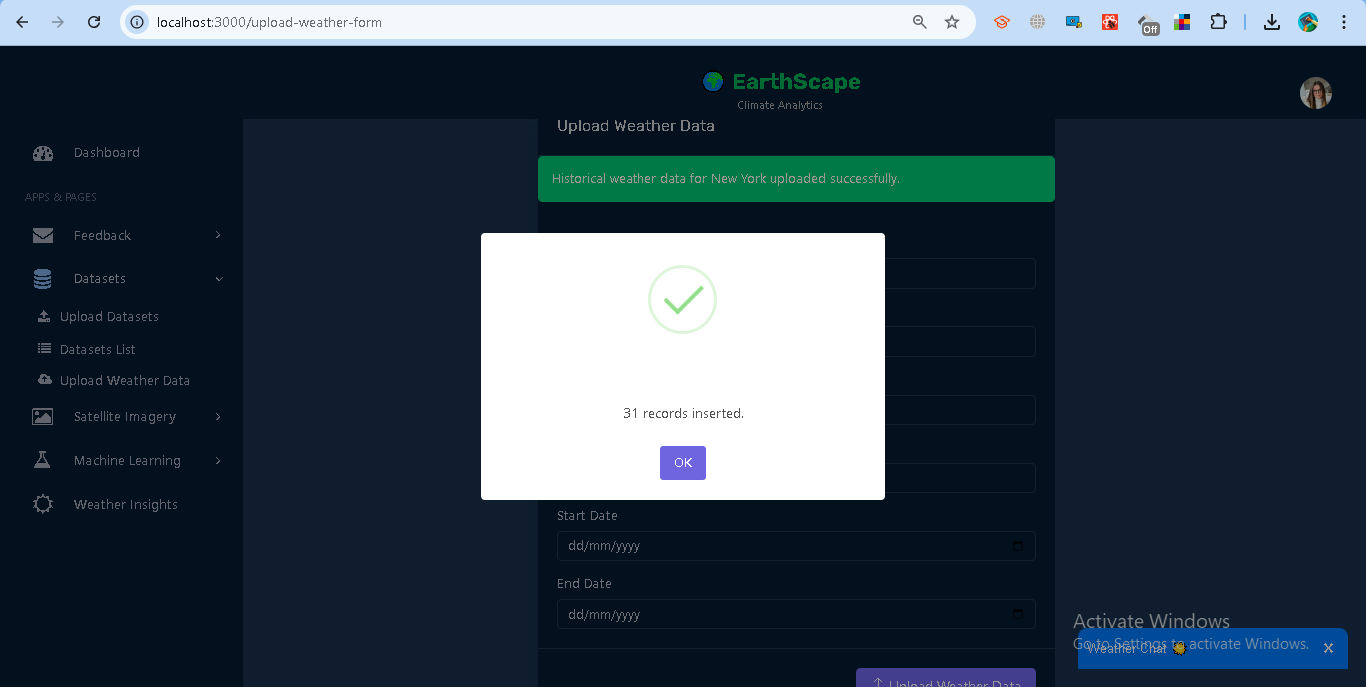
            "csv\_file": filename

        }), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

1. The system queries Meteostat for daily climate records.
2. Records include: temperature, humidity, pressure, wind speed, and precipitation.
3. A .csv file with the results is stored and available for download.
4. Confirmation message appears:



Code:

import React, { useState, useEffect } from "react";

import { useNavigate } from "react-router-dom";

import axios from "axios";

import Header from "../dashboard/Header";

import Sidebar from "../dashboard/Sidebar";

import Footer from "../dashboard/Footer";

import Swal from "sweetalert2";

const OPENCAGE\_API\_KEY = "b3d392b8ecb14efc808b3006158fa3e0";

function UploadWeatherForm() {

  const [datasetName, setDatasetName] = useState("");

  const [city, setCity] = useState("");

  const [latitude, setLatitude] = useState("");

  const [longitude, setLongitude] = useState("");

  const [startDate, setStartDate] = useState("");

  const [endDate, setEndDate] = useState("");

  const [message, setMessage] = useState("");

  const [error, setError] = useState("");

  const navigate = useNavigate();

  // Redirect if not logged in

  useEffect(() => {

    const token = localStorage.getItem("token");

    if (!token) {

      Swal.fire({

        icon: "warning",

        title: "Login Required",

        text: "Please log in to access this page.",

        confirmButtonText: "OK",

      }).then(() => navigate("/login"));

    }

  }, [navigate]);

  // Fetch coordinates

  const fetchCoordinates = async (cityName) => {

    try {

      if (!cityName) return;

      const res = await axios.get(

        `https://api.opencagedata.com/geocode/v1/json?q=${encodeURIComponent(

          cityName

        )}&key=${OPENCAGE\_API\_KEY}`

      );

      if (res.data && res.data.results.length > 0) {

        const { lat, lng } = res.data.results[0].geometry;

        setLatitude(lat);

        setLongitude(lng);

        setError("");

      } else {

        setError("City not found. Please enter a valid city.");

      }

    } catch (err) {

      setError("Failed to fetch coordinates. Try again later.");

    }

  };

  const handleSubmit = async (e) => {

    e.preventDefault();

    if (!datasetName || !city || !latitude || !longitude || !startDate || !endDate) {

      setError("Please fill in all fields.");

      setMessage("");

      return;

    }

    try {

      const token = localStorage.getItem("token");

      if (!token) {

        setError("You must be logged in.");

        return;

      }

      const res = await axios.post(

        "http://127.0.0.1:5000/api/upload-weather",

        {

          dataset\_name: datasetName,

          city,

          latitude,

          longitude,

          start\_date: startDate,

          end\_date: endDate,

        },

        {

          headers: {

            Authorization: `Bearer ${token}`,

          },

        }

      );

      setMessage(res.data.message || "Weather data uploaded successfully!");

      setError("");

      setDatasetName("");

      setCity("");

      setLatitude("");

      setLongitude("");

      setStartDate("");

      setEndDate("");

      Swal.fire({

        icon: "success",

        title: "Uploaded!",

        text: `${res.data.records\_count} records inserted.`,

      });

    } catch (err) {

      setError(err.response?.data?.error || "An error occurred while uploading weather data.");

      setMessage("");

    }

  };

  return (

    <div className="wrapper">

      <Header />

      <Sidebar />

      <div

        className="content-wrapper d-flex justify-content-center align-items-center"

        style={{ minHeight: "80vh", padding: "20px" }}

      >

        <div className="col-lg-6 col-12">

          <div className="box">

            <div className="box-header with-border">

              <h4 className="box-title">Upload Weather Data</h4>

            </div>

            {message && <div className="alert alert-success">{message}</div>}

            {error && <div className="alert alert-danger">{error}</div>}

            <form className="form" onSubmit={handleSubmit}>

              <div className="box-body">

                <div className="form-group">

                  <label className="form-label">Dataset Name</label>

                  <input

                    type="text"

                    className="form-control"

                    value={datasetName}

                    onChange={(e) => setDatasetName(e.target.value)}

                    placeholder="Enter Dataset Name"

                  />

                </div>

                <div className="form-group">

                  <label className="form-label">City</label>

                  <input

                    type="text"

                    className="form-control"

                    value={city}

                    onChange={(e) => {

                      setCity(e.target.value);

                      fetchCoordinates(e.target.value);

                    }}

                    placeholder="Enter City Name"

                  />

                </div>

                <div className="form-group">

                  <label className="form-label">Latitude</label>

                  <input type="text" className="form-control" value={latitude} disabled readOnly />

                </div>

                <div className="form-group">

                  <label className="form-label">Longitude</label>

                  <input type="text" className="form-control" value={longitude} disabled readOnly />

                </div>

                <div className="form-group">

                  <label className="form-label">Start Date</label>

                  <input

                    type="date"

                    className="form-control"

                    value={startDate}

                    onChange={(e) => setStartDate(e.target.value)}

                  />

                </div>

                <div className="form-group">

                  <label className="form-label">End Date</label>

                  <input

                    type="date"

                    className="form-control"

                    value={endDate}

                    onChange={(e) => setEndDate(e.target.value)}

                  />

                </div>

              </div>

              <div className="box-footer text-end">

                <button type="submit" className="btn btn-primary">

                  <i className="ti-upload"></i> Upload Weather Data

                </button>

              </div>

            </form>

          </div>

        </div>

      </div>

      <Footer />

    </div>

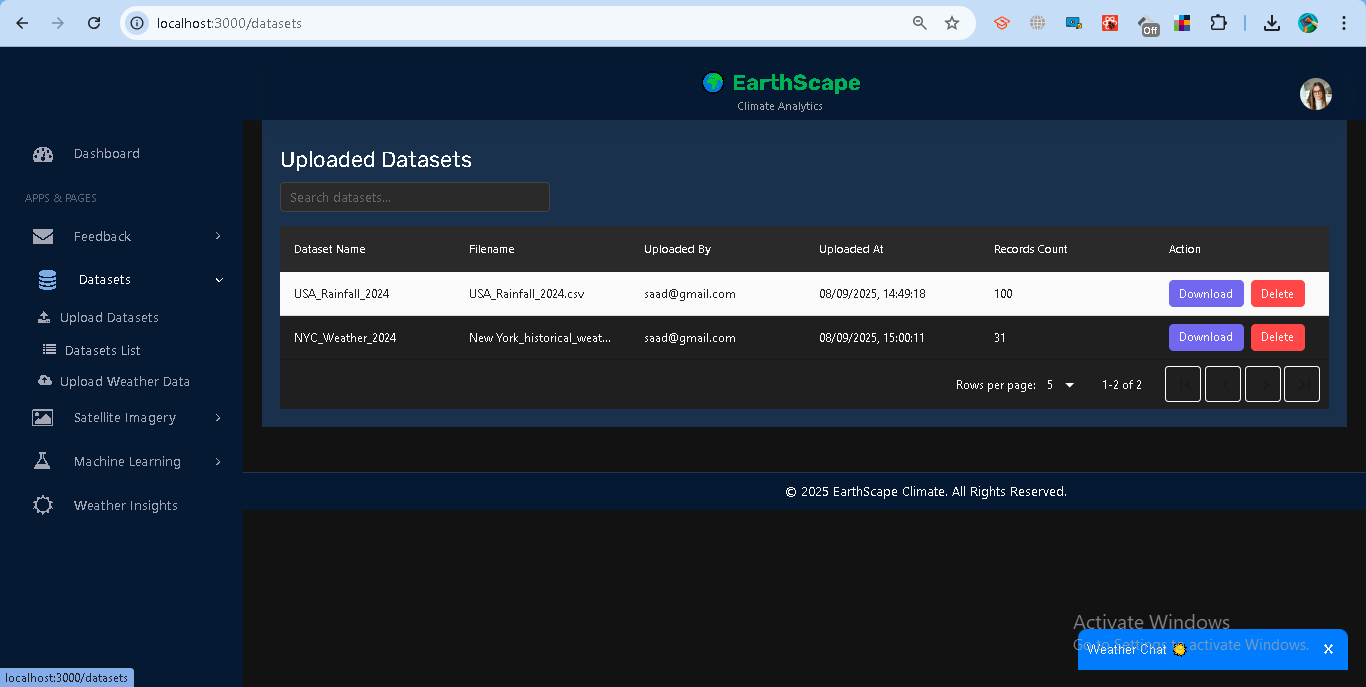
  );

}

export default UploadWeatherForm;

### **Dataset Management**

* **View Datasets:** All uploaded datasets can be viewed under **Dataset → *Datasets lists*.**

****

Code:

import React, { useEffect, useState } from "react";

import axios from "axios";

import Swal from "sweetalert2";

import DataTable from "react-data-table-component";

import Header from "../dashboard/Header";

import Sidebar from "../dashboard/Sidebar";

import Footer from "../dashboard/Footer";

function DatasetList() {

  const [datasets, setDatasets] = useState([]);

  const [filteredDatasets, setFilteredDatasets] = useState([]);

  const [searchText, setSearchText] = useState("");

  const [loading, setLoading] = useState(true);

  useEffect(() => {

    const token = localStorage.getItem("token");

    if (!token) {

      Swal.fire({

        icon: "warning",

        title: "Login Required",

        text: "Please log in first.",

      });

      return;

    }

    fetchDatasets(token);

  }, []);

  const fetchDatasets = async (token) => {

    try {

      const res = await axios.get("http://127.0.0.1:5000/api/datasets", {

        headers: { Authorization: `Bearer ${token}` },

      });

      // Flatten nested datasets for table

      const flat = [];

      res.data.datasets.forEach((userDoc) => {

        userDoc.datasets.forEach((ds) => {

          flat.push({

            doc\_id: userDoc.\_id,   // ✅ actual MongoDB ObjectId for parent doc

            user\_id: userDoc.user\_id,

            dataset\_name: ds.dataset\_name,

            filename: ds.filename,

            uploaded\_at: ds.uploaded\_at,

            uploaded\_by: ds.uploaded\_by,

            records\_count: ds.records\_count,

            download\_url: ds.download\_url,

          });

        });

      });

      setDatasets(flat);

      setFilteredDatasets(flat);

      setLoading(false);

    } catch (err) {

      Swal.fire({

        icon: "error",

        title: "Error",

        text: err.response?.data?.error || "Failed to load datasets.",

      });

      setLoading(false);

    }

  };

  // 🗑️ Delete dataset

  const handleDelete = async (docId, filename) => {

    const token = localStorage.getItem("token");

    if (!token) return;

    Swal.fire({

      title: "Are you sure?",

      text: `Delete dataset "${filename}"?`,

      icon: "warning",

      showCancelButton: true,

      confirmButtonColor: "#d33",

      cancelButtonColor: "#3085d6",

      confirmButtonText: "Yes, delete it!",

    }).then(async (result) => {

      if (result.isConfirmed) {

        try {

          await axios.delete(

            `http://127.0.0.1:5000/api/datasets/${docId}/${filename}`,

            { headers: { Authorization: `Bearer ${token}` } }

          );

          Swal.fire("Deleted!", "Dataset has been deleted.", "success");

          // Remove deleted dataset from state

          const updated = datasets.filter(

            (d) => !(d.doc\_id === docId && d.filename === filename)

          );

          setDatasets(updated);

          setFilteredDatasets(updated);

        } catch (err) {

          Swal.fire({

            icon: "error",

            title: "Error",

            text: err.response?.data?.error || "Failed to delete dataset.",

          });

        }

      }

    });

  };

  useEffect(() => {

    const lower = searchText.toLowerCase();

    const filtered = datasets.filter(

      (d) =>

        d.dataset\_name.toLowerCase().includes(lower) ||

        d.filename.toLowerCase().includes(lower) ||

        d.uploaded\_by.toLowerCase().includes(lower)

    );

    setFilteredDatasets(filtered);

  }, [searchText, datasets]);

  const columns = [

    { name: "Dataset Name", selector: (row) => row.dataset\_name, sortable: true },

    { name: "Filename", selector: (row) => row.filename, sortable: true },

    { name: "Uploaded By", selector: (row) => row.uploaded\_by, sortable: true },

    {

      name: "Uploaded At",

      selector: (row) => row.uploaded\_at,

      sortable: true,

      cell: (row) =>

        row.uploaded\_at ? new Date(row.uploaded\_at).toLocaleString() : "N/A",

    },

    { name: "Records Count", selector: (row) => row.records\_count, sortable: true },

    {

      name: "Action",

      cell: (row) => (

        <div style={{ display: "flex", gap: "8px" }}>

          <a

            href={row.download\_url}

            className="btn btn-sm btn-primary"

            target="\_blank"

            rel="noopener noreferrer"

          >

            Download

          </a>

          <button

            className="btn btn-sm btn-danger"

            onClick={() => handleDelete(row.doc\_id, row.filename)} // ✅ use doc\_id

          >

            Delete

          </button>

        </div>

      ),

    },

  ];

  const darkTableStyles = {

    header: { style: { backgroundColor: "#1f1f1f", color: "#fff" } },

    headRow: { style: { backgroundColor: "#2a2a2a", color: "#fff" } },

    headCells: { style: { color: "#fff" } },

    rows: {

      style: { backgroundColor: "#1f1f1f", color: "#fff" },

      highlightOnHoverStyle: { backgroundColor: "#333", color: "#fff" },

    },

    pagination: {

      style: { backgroundColor: "#1f1f1f", color: "#fff" },

      pageButtonsStyle: {

        color: "#fff",

        fill: "#fff",

        backgroundColor: "transparent",

        border: "1px solid #fff",

        borderRadius: "4px",

        padding: "4px 10px",

        cursor: "pointer",

        margin: "0 2px",

      },

    },

  };

  return (

    <div className="wrapper" style={{ backgroundColor: "#121212", minHeight: "100vh" }}>

      <Header />

      <Sidebar />

      <div className="content-wrapper" style={{ padding: "20px" }}>

        <h3 style={{ color: "#fff" }}>Uploaded Datasets</h3>

        <input

          type="text"

          placeholder="Search datasets..."

          value={searchText}

          onChange={(e) => setSearchText(e.target.value)}

          className="form-control"

          style={{

            maxWidth: "300px",

            marginBottom: "15px",

            backgroundColor: "#2a2a2a",

            color: "#fff",

            border: "1px solid #444",

          }}

        />

        {loading ? (

          <p style={{ color: "#fff" }}>Loading...</p>

        ) : (

          <DataTable

            columns={columns}

            data={filteredDatasets}

            pagination

            paginationPerPage={5}

            paginationRowsPerPageOptions={[5, 10, 15]}

            highlightOnHover

            striped

            responsive

            customStyles={darkTableStyles}

          />

        )}

      </div>

      <Footer />

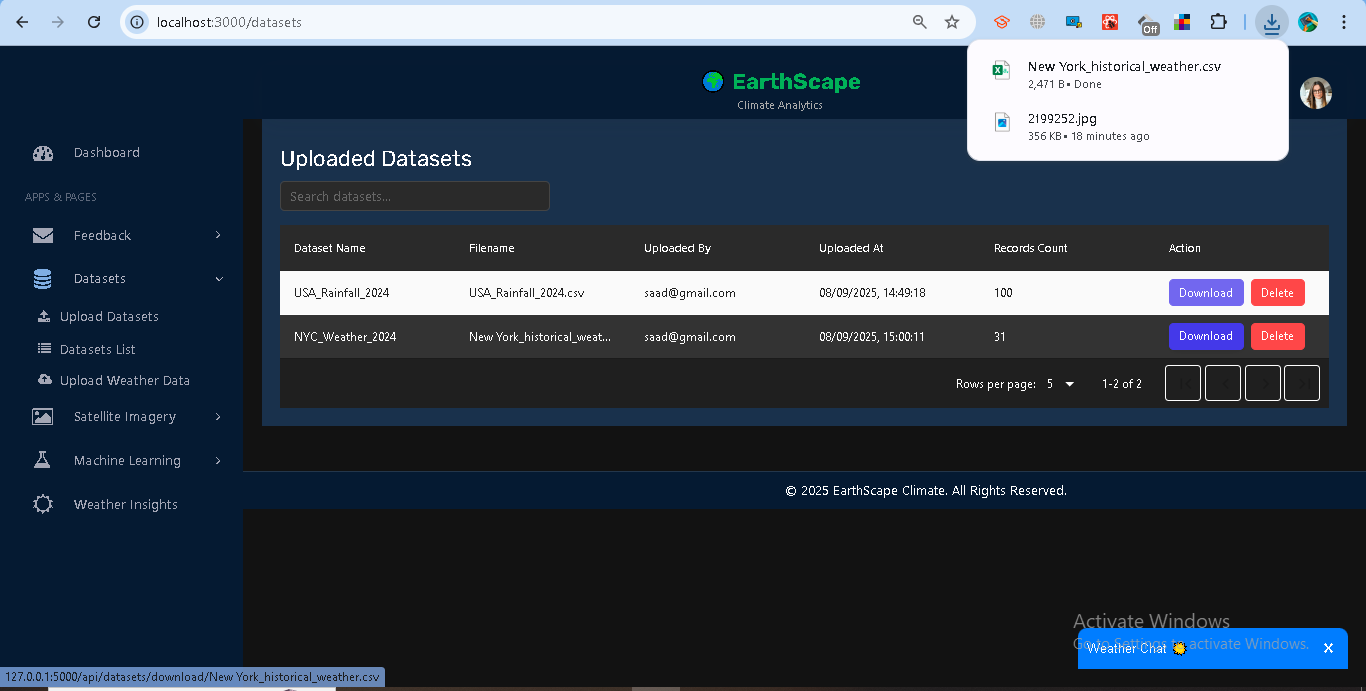
    </div>

  );

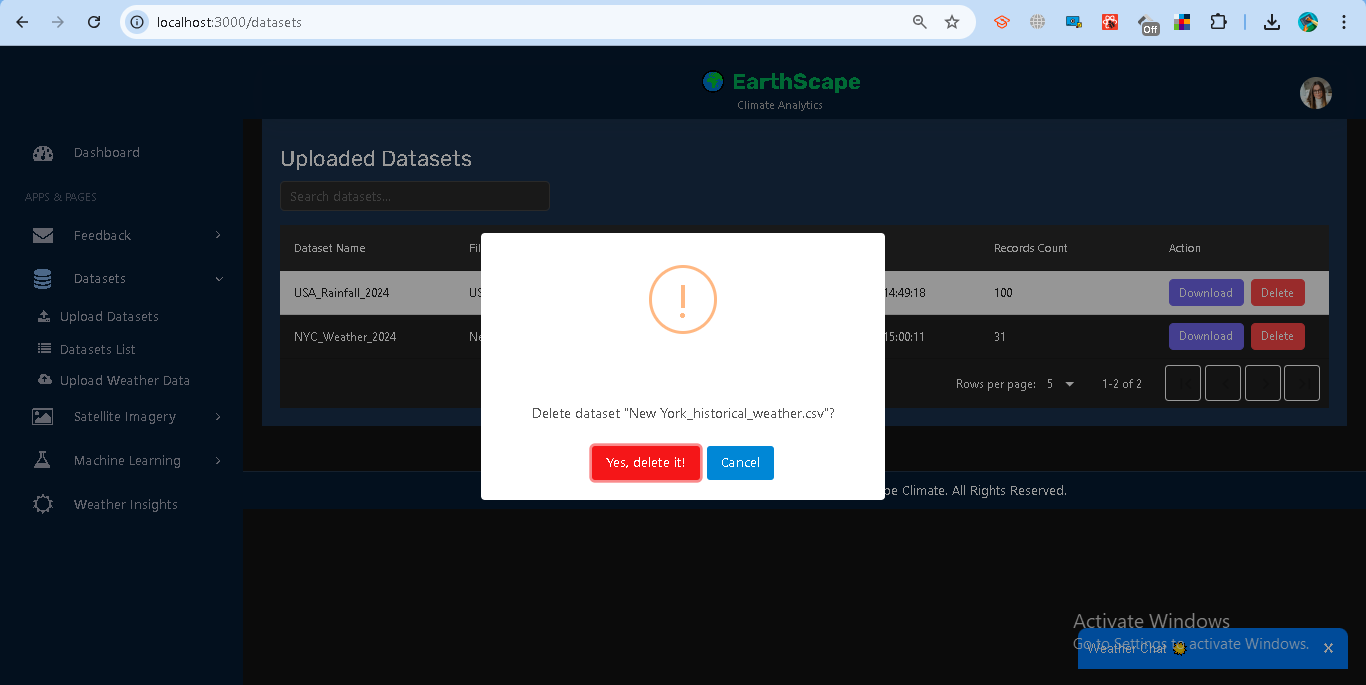
}

export default DatasetList;

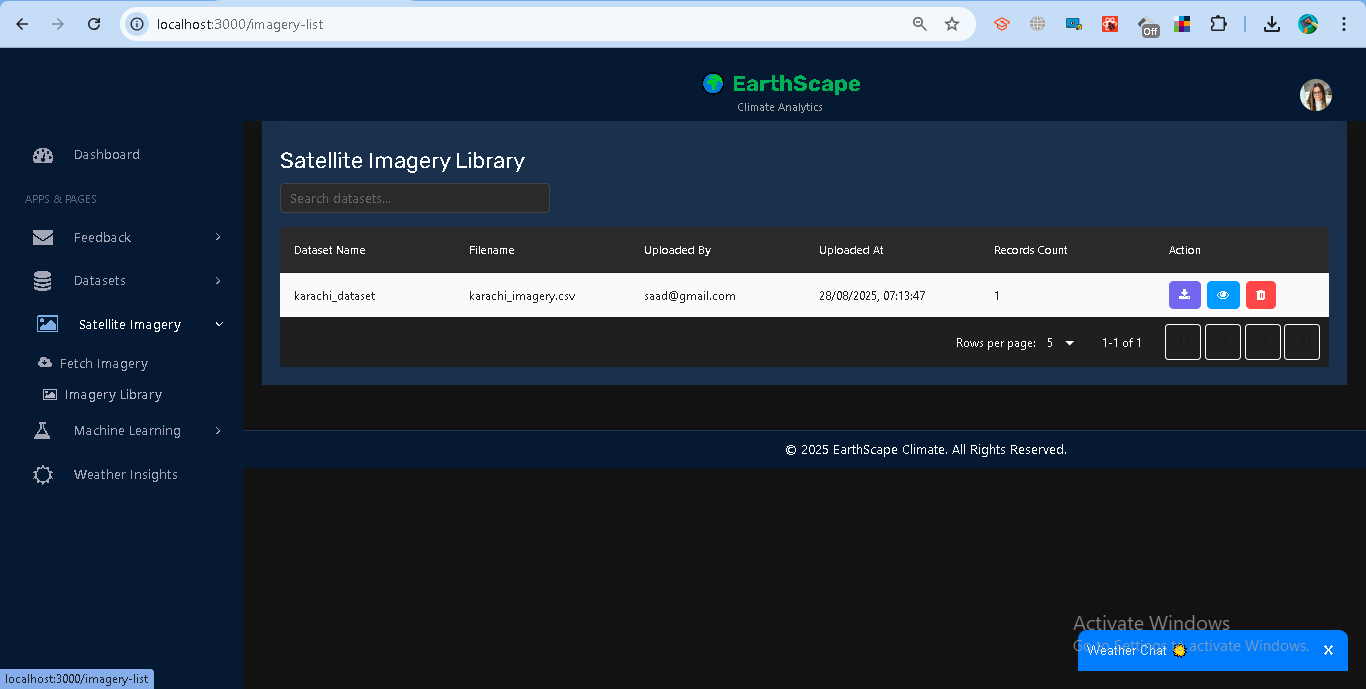
* **Download:** Each dataset provides a direct download link for .csv files.

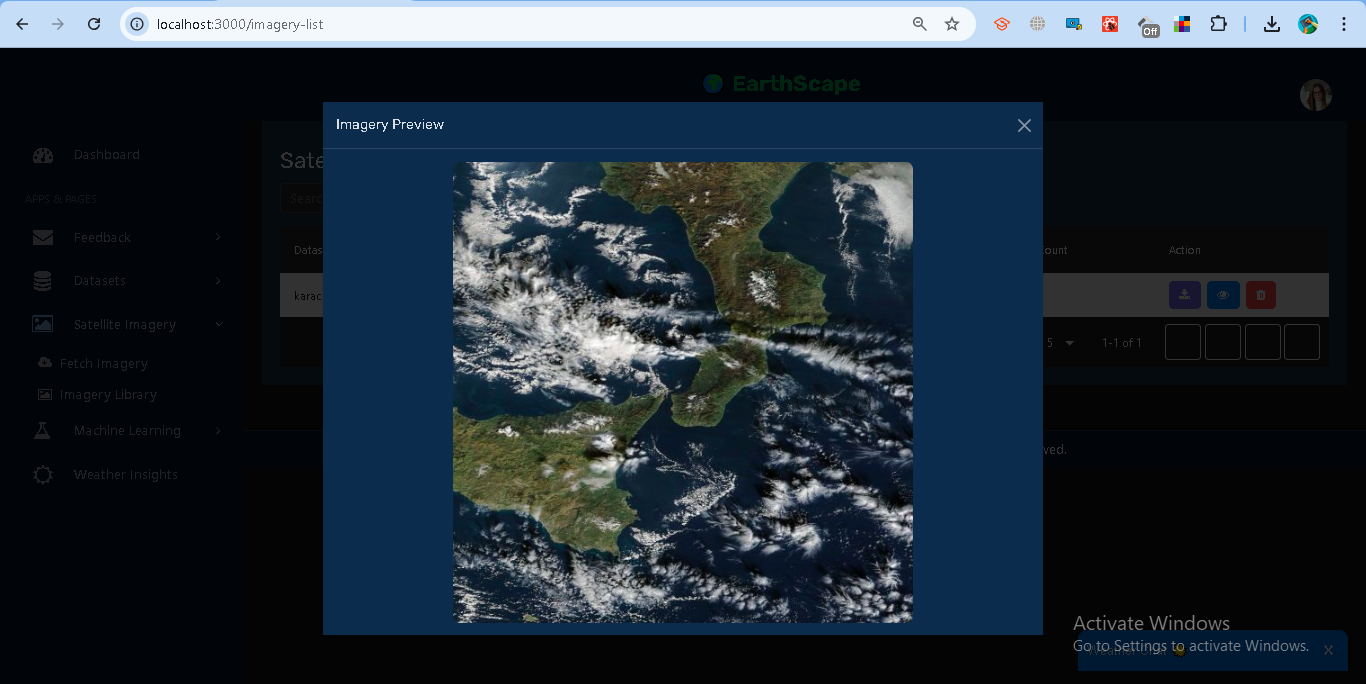


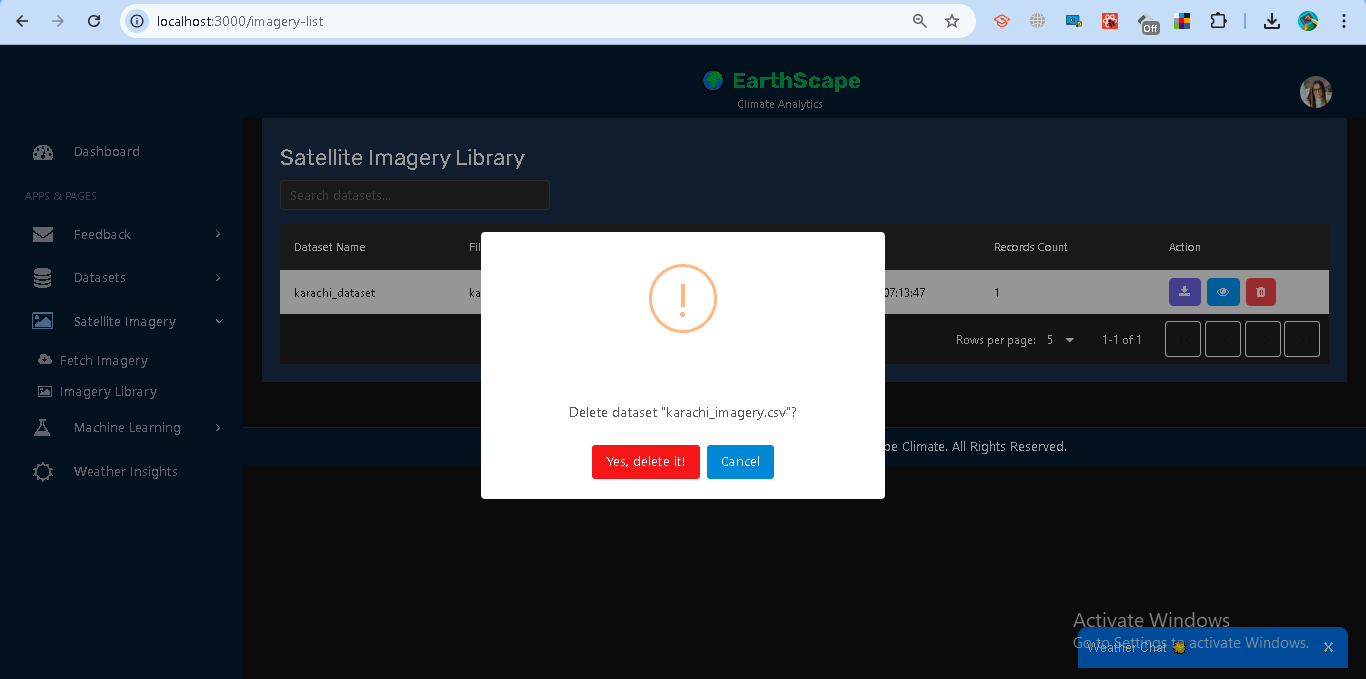
* **Delete:** Users may delete their own datasets; admins can delete all datasets.



* **View Datasets:** All uploaded datasets can be viewed under **Satellite Imagery → *Imagery Library*.**

****

****

****

## **1.3 Data Storage**

### **Implementation**

Instead of using Hadoop Distributed File System (HDFS), this project leverages **MongoDB Atlas** for storing climate-related datasets in a **scalable and fault-tolerant** way. MongoDB’s document-oriented design provides flexibility for handling diverse climate datasets such as CSV uploads, satellite imagery metadata, and weather station records.

* **Datasets Collection (datasets)**
  + Stores uploaded CSV datasets along with metadata (dataset name, file path, uploader, records, record counts).

Example structure:  
  
 {

"user\_id": "65ab3e...",

"datasets": [

{

"dataset\_name": "NYC\_Weather\_2024",

"filename": "nyc\_weather.csv",

"file\_path": "/dataset/nyc\_weather.csv",

"uploaded\_at": "2025-01-12T08:30:00Z",

"uploaded\_by": "admin@example.com",

"records\_count": 100,

"records": [ { "date": "2024-01-01", "temperature": 5, "humidity": 80 } ]

}

]

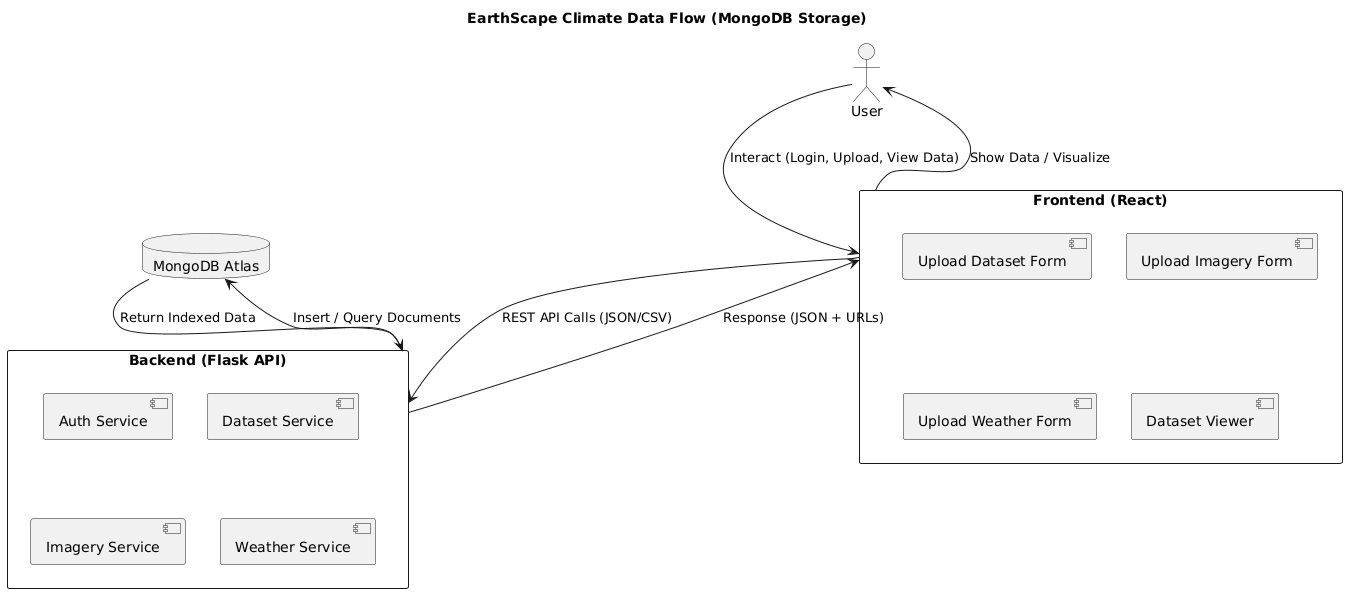
}

* **Satellite Imagery Collection (satellite\_imagery)**
  + Stores metadata about imagery requests (city, lat/lon, zoom level, NASA GIBS URL, generated CSV file, etc.).
* **User Profile Collection (users)**
  + Stores user credentials, roles, and cities, enabling role-based access control.

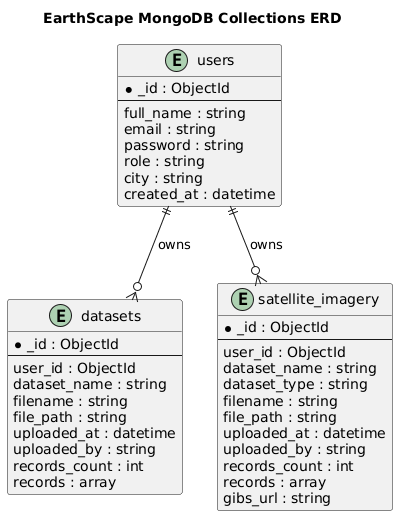
### **Optimization Strategies**

* **Indexing**
  + MongoDB indexes are created on fields like user\_id, datasets.filename, and satellite\_imagery.dataset\_name to speed up queries.
* **Partitioning / Sharding**
  + MongoDB supports **sharding** for horizontal scaling when datasets grow large. This ensures queries and storage are distributed across multiple nodes.
* **Document Nesting**
  + Datasets are stored as **nested arrays under each user document**, ensuring quick retrieval of all datasets for a given user.
* **File Storage**
  + Uploaded CSVs and imagery metadata are stored locally under /dataset and referenced in MongoDB with **download URLs** for retrieval.

**Data Flow**

****

**ERD**

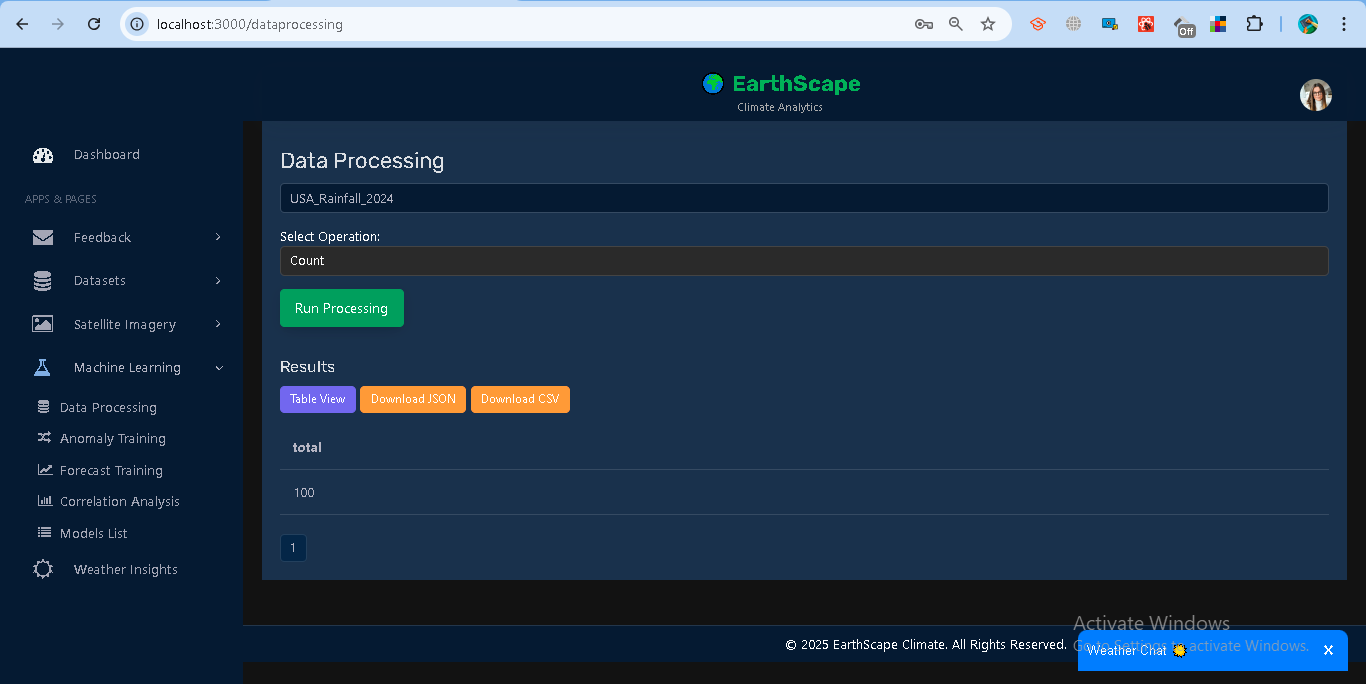
****

### **1.4 Data Processing**

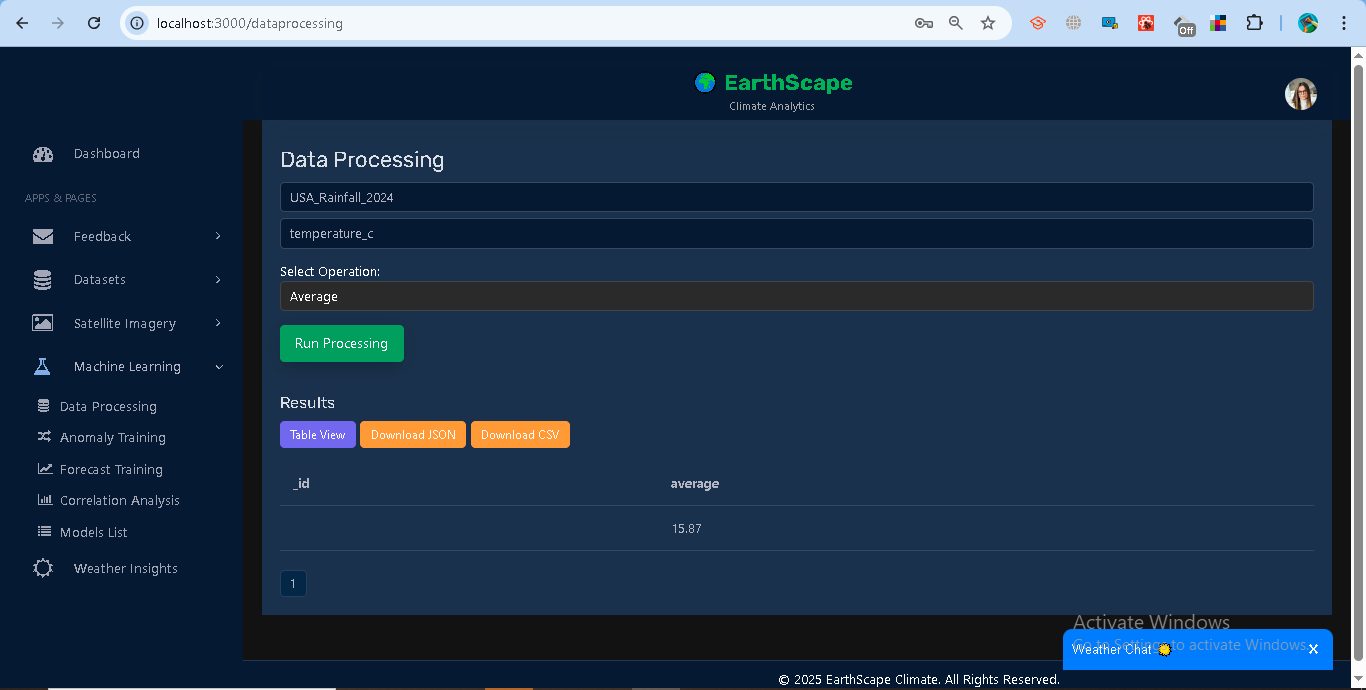
**Purpose** Allows users to analyze climate datasets (CSV, Weather, Satellite) stored in MongoDB. Operations include record counts, averages, groupings, and simple temporal pattern detection.

**Available Operations**

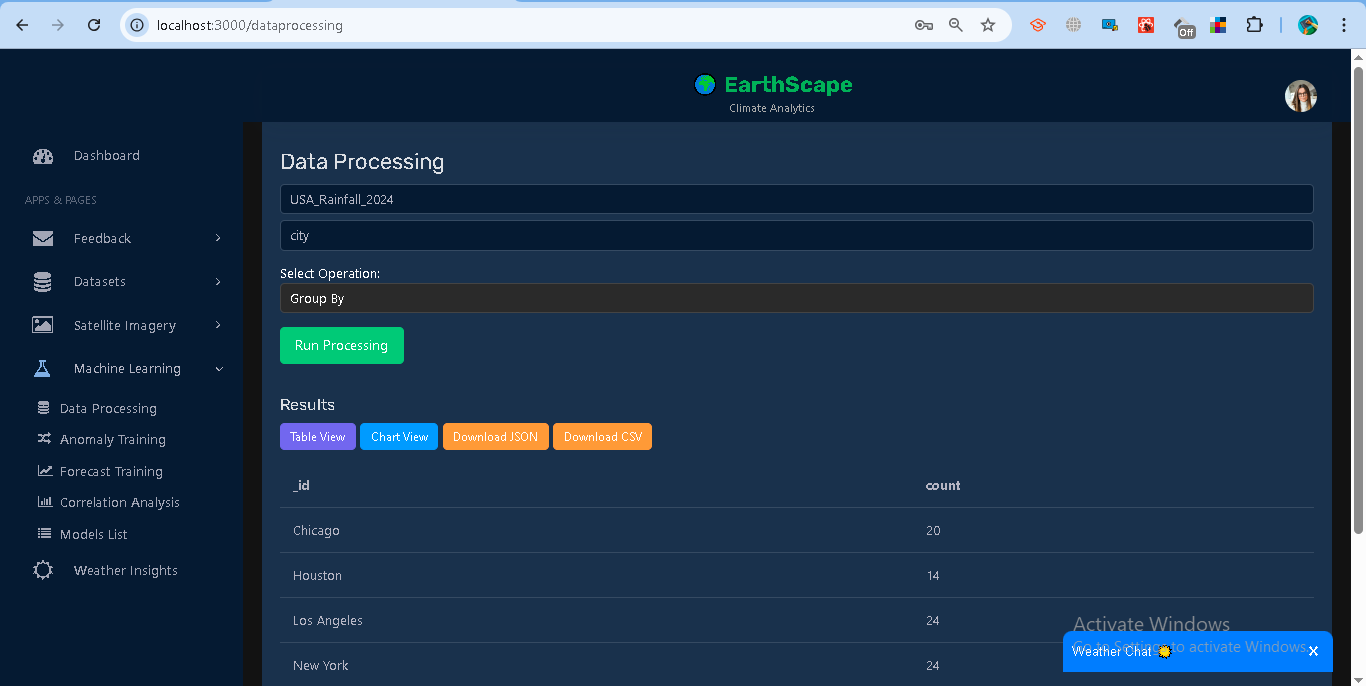
* **Count** → total records.



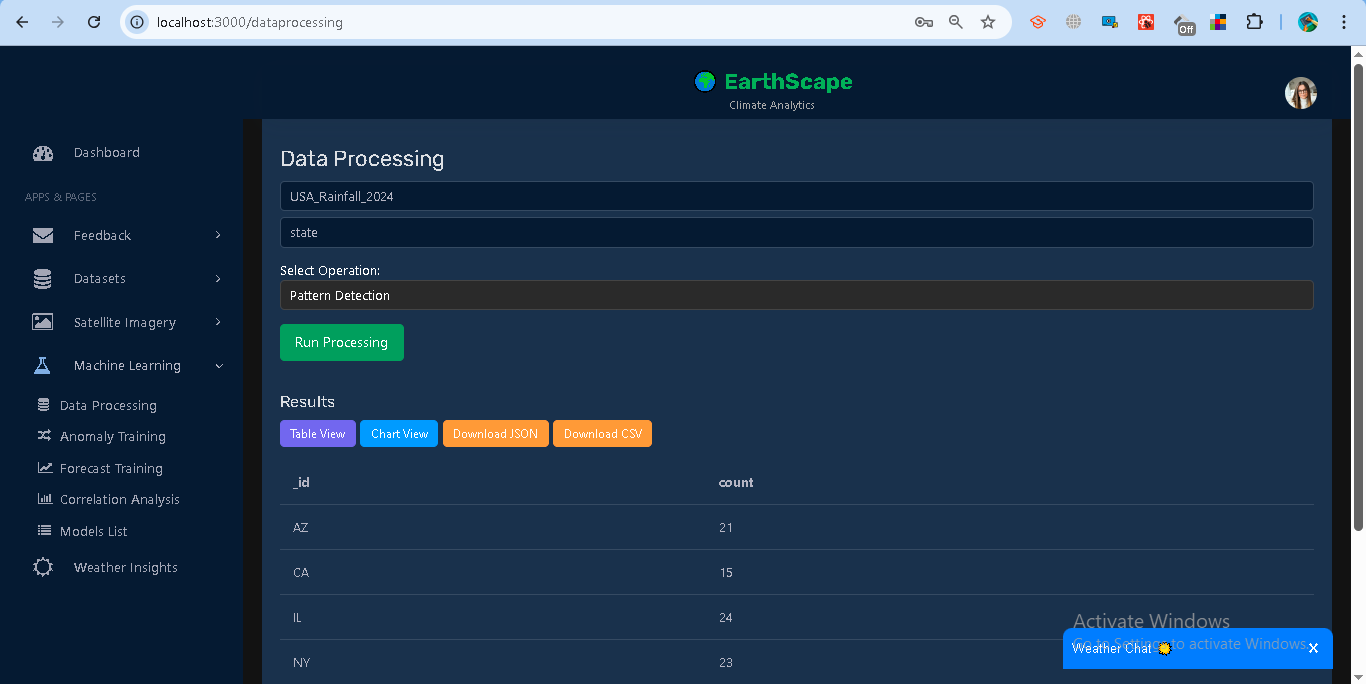
* **Average** → mean of numeric field.



* **Group By** → group by field (city, value bins).



* **Pattern** → detect seasonality (group by month if date).

  
Code:

import React, { useEffect, useState } from "react";

import axios from "axios";

import Swal from "sweetalert2";

import Header from "../dashboard/Header";

import Sidebar from "../dashboard/Sidebar";

import Footer from "../dashboard/Footer";

import {

  BarChart,

  Bar,

  XAxis,

  YAxis,

  Tooltip,

  ResponsiveContainer,

  LineChart,

  Line,

} from "recharts";

function DataProcessing() {

  const [datasets, setDatasets] = useState([]);

  const [selectedDataset, setSelectedDataset] = useState("");

  const [fields, setFields] = useState([]);

  const [selectedField, setSelectedField] = useState("");

  const [operation, setOperation] = useState("count");

  const [results, setResults] = useState(null);

  const [view, setView] = useState("table"); // table | chart

  const [page, setPage] = useState(1);

  const rowsPerPage = 20;

  const monthNames = [

    "Jan","Feb","Mar","Apr","May","Jun",

    "Jul","Aug","Sep","Oct","Nov","Dec"

  ];

  useEffect(() => {

    const token = localStorage.getItem("token");

    if (!token) {

      Swal.fire({

        icon: "warning",

        title: "Login Required",

        text: "Please log in first.",

      });

      return;

    }

    fetchDatasets(token);

  }, []);

  const fetchDatasets = async (token) => {

    try {

      const res = await axios.get("http://127.0.0.1:5000/api/datasets", {

        headers: { Authorization: `Bearer ${token}` },

      });

      // Flatten nested datasets

      const flat = [];

      res.data.datasets.forEach((userDoc) => {

        userDoc.datasets.forEach((ds) => {

          flat.push({

            doc\_id: userDoc.\_id,

            dataset\_name: ds.dataset\_name,

            filename: ds.filename,

          });

        });

      });

      setDatasets(flat);

    } catch (err) {

      Swal.fire({

        icon: "error",

        title: "Error",

        text: err.response?.data?.error || "Failed to load datasets.",

      });

    }

  };

  const fetchFields = async (datasetName) => {

    setSelectedDataset(datasetName);

    try {

      const token = localStorage.getItem("token");

      const res = await axios.get(

        `http://127.0.0.1:5000/api/dataset/${datasetName}/fields`,

        { headers: { Authorization: `Bearer ${token}` } }

      );

      setFields(res.data.fields);

    } catch (err) {

      Swal.fire({

        icon: "error",

        title: "Error",

        text: err.response?.data?.error || "Failed to fetch fields.",

      });

    }

  };

  const runProcessing = async () => {

    if (!selectedDataset || !operation) {

      Swal.fire("Error", "Please select dataset and operation.", "warning");

      return;

    }

  if (["average", "group\_by", "pattern"].includes(operation) && !selectedField) {

    Swal.fire("Error", "Please select a field for this operation.", "warning");

    return;

  }

    try {

      const token = localStorage.getItem("token");

      const res = await axios.post(

        `http://127.0.0.1:5000/api/dataset/${selectedDataset}/process`,

        { operation, field: selectedField },

        { headers: { Authorization: `Bearer ${token}` } }

      );

      setResults(res.data);

      setPage(1); // reset pagination

    } catch (err) {

      Swal.fire({

        icon: "error",

        title: "Error",

        text: err.response?.data?.error || "Failed to process dataset.",

      });

    }

  };

  const downloadResults = (type = "json") => {

    if (!results) return;

    if (type === "json") {

      const blob = new Blob([JSON.stringify(results, null, 2)], {

        type: "application/json",

      });

      const url = URL.createObjectURL(blob);

      const link = document.createElement("a");

      link.href = url;

      link.download = "results.json";

      link.click();

    } else if (type === "csv") {

      const keys = Object.keys(results[0] || {});

      const csv = [

        keys.join(","),

        ...results.map((row) => keys.map((k) => row[k]).join(",")),

      ].join("\n");

      const blob = new Blob([csv], { type: "text/csv" });

      const url = URL.createObjectURL(blob);

      const link = document.createElement("a");

      link.href = url;

      link.download = "results.csv";

      link.click();

    }

  };

  // Pagination

  const paginatedResults = Array.isArray(results)

    ? results.slice((page - 1) \* rowsPerPage, page \* rowsPerPage)

    : results;

  // Top 20 for charts

  const topResults =

    (operation === "group\_by" || operation === "pattern") && Array.isArray(results)

      ? [...results].sort((a, b) => b.count - a.count).slice(0, 20)

      : [];

  return (

    <div className="wrapper" style={{ backgroundColor: "#121212", minHeight: "100vh" }}>

      <Header />

      <Sidebar />

      <div className="content-wrapper" style={{ padding: "20px", color: "#fff" }}>

        <h3>Data Processing</h3>

        {/\* Dataset Dropdown \*/}

        <select

          className="form-control"

          value={selectedDataset}

          onChange={(e) => fetchFields(e.target.value)}

        >

          <option value="">-- Choose Dataset --</option>

          {datasets.map((d, idx) => (

            <option key={idx} value={d.dataset\_name}>

              {d.dataset\_name}

            </option>

          ))}

        </select>

        {/\* Fields Dropdown \*/}

      {fields.length > 0 && operation !== "count" && (

  <select

    className="form-control mt-2"

    value={selectedField}

    onChange={(e) => setSelectedField(e.target.value)}

  >

    <option value="">-- Choose Field --</option>

    {fields.map((f, idx) => (

      <option key={idx} value={f}>

        {f}

      </option>

    ))}

  </select>

)}

        {/\* Operation Dropdown \*/}

        <div style={{ margin: "15px 0" }}>

          <label>Select Operation:</label>

          <select

            className="form-control"

            value={operation}

            onChange={(e) => setOperation(e.target.value)}

            style={{ backgroundColor: "#2a2a2a", color: "#fff", border: "1px solid #444" }}

          >

            <option value="count">Count</option>

            <option value="average">Average</option>

            <option value="group\_by">Group By</option>

            <option value="pattern">Pattern Detection</option>

          </select>

        </div>

        <button

  className="btn btn-success mb-3"

  onClick={runProcessing}

  disabled={

    !selectedDataset ||

    (["average", "group\_by", "pattern"].includes(operation) && !selectedField)

  }

>

  Run Processing

</button>

        {/\* Results \*/}

        {results && (

          <div style={{ marginTop: "20px" }}>

            <h4>Results</h4>

            {/\* Controls \*/}

            <div className="mb-3">

              <button

                className="btn btn-sm btn-primary me-5"

                onClick={() => setView("table")}

              >

                Table View

              </button>

              {(operation === "group\_by" || operation === "pattern") && (

                <button

                  className="btn btn-sm btn-info me-5"

                  onClick={() => setView("chart")}

                >

                  Chart View

                </button>

              )}

              <button

                className="btn btn-sm btn-warning me-5"

                onClick={() => downloadResults("json")}

              >

                Download JSON

              </button>

              {Array.isArray(results) && (

                <button

                  className="btn btn-sm btn-warning"

                  onClick={() => downloadResults("csv")}

                >

                  Download CSV

                </button>

              )}

            </div>

            {/\* Table View \*/}

            {view === "table" && Array.isArray(results) && (

              <div>

                <div style={{ maxHeight: "400px", overflowY: "auto" }}>

                  <table className="table table-dark table-sm">

                    <thead>

                      <tr>

                        {Object.keys(results[0] || {}).map((k) => (

                          <th key={k}>{k}</th>

                        ))}

                      </tr>

                    </thead>

                    <tbody>

                      {paginatedResults.map((row, idx) => (

                        <tr key={idx}>

                          {Object.values(row).map((v, i) => (

                            <td key={i}>{v}</td>

                          ))}

                        </tr>

                      ))}

                    </tbody>

                  </table>

                </div>

                {/\* Pagination \*/}

                <div className="mt-2">

                  {Array.from({

                    length: Math.ceil(results.length / rowsPerPage),

                  }).map((\_, i) => (

                    <button

                      key={i}

                      className={`btn btn-sm me-5 ${

                        page === i + 1 ? "btn-light" : "btn-secondary"

                      } mr-1`}

                      onClick={() => setPage(i + 1)}

                    >

                      {i + 1}

                    </button>

                  ))}

                </div>

              </div>

            )}

            {/\* Single object result \*/}

            {view === "table" && !Array.isArray(results) && (

              <pre

                style={{

                  backgroundColor: "#2a2a2a",

                  padding: "10px",

                  borderRadius: "5px",

                  color: "#fff",

                }}

              >

                {JSON.stringify(results, null, 2)}

              </pre>

            )}

            {/\* Chart View \*/}

            {view === "chart" &&

              (operation === "group\_by" || operation === "pattern") &&

              topResults.length > 0 && (

                <ResponsiveContainer width="100%" height={350}>

  {operation === "group\_by" ? (

    // Always bar chart for group\_by

    <BarChart data={topResults}>

      <XAxis dataKey="\_id" stroke="#fff" />

      <YAxis stroke="#fff" />

      <Tooltip />

      <Bar dataKey="count" fill="#3b82f6" />

    </BarChart>

  ) : operation === "pattern" ? (

    // Pattern → decide chart type based on field values

    Number.isInteger(topResults[0]?.\_id) &&

    topResults[0].\_id >= 1 &&

    topResults[0].\_id <= 12 ? (

      // If \_id looks like a month → LineChart

      <LineChart data={topResults}>

        <XAxis

          dataKey="\_id"

          stroke="#fff"

          tickFormatter={(val) => monthNames[val - 1] || val}

        />

        <YAxis stroke="#fff" />

        <Tooltip labelFormatter={(val) => monthNames[val - 1] || val} />

        <Line type="monotone" dataKey="count" stroke="#3b82f6" />

      </LineChart>

    ) : (

      // Otherwise numeric/string pattern → BarChart

      <BarChart data={topResults}>

        <XAxis dataKey="\_id" stroke="#fff" />

        <YAxis stroke="#fff" />

        <Tooltip />

        <Bar dataKey="count" fill="#3b82f6" />

      </BarChart>

    )

  ) : null}

</ResponsiveContainer>

              )}

          </div>

        )}

      </div>

      <Footer />

    </div>

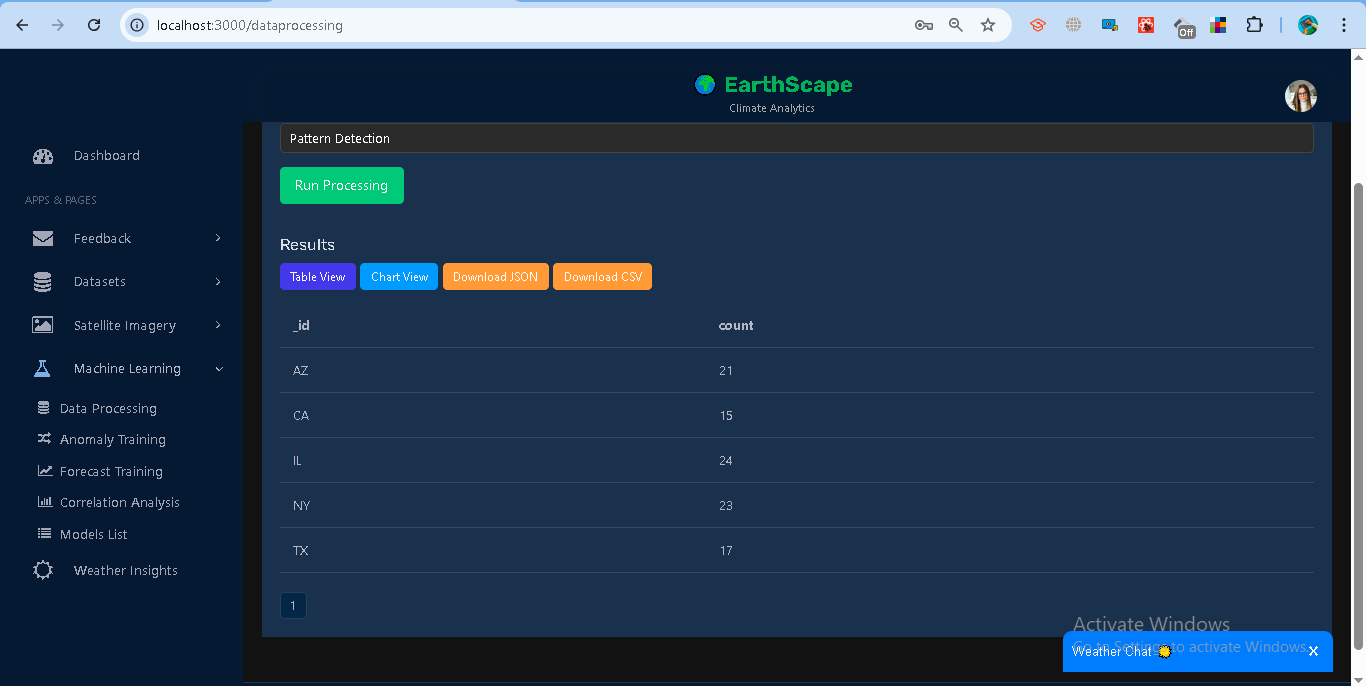
  );

}

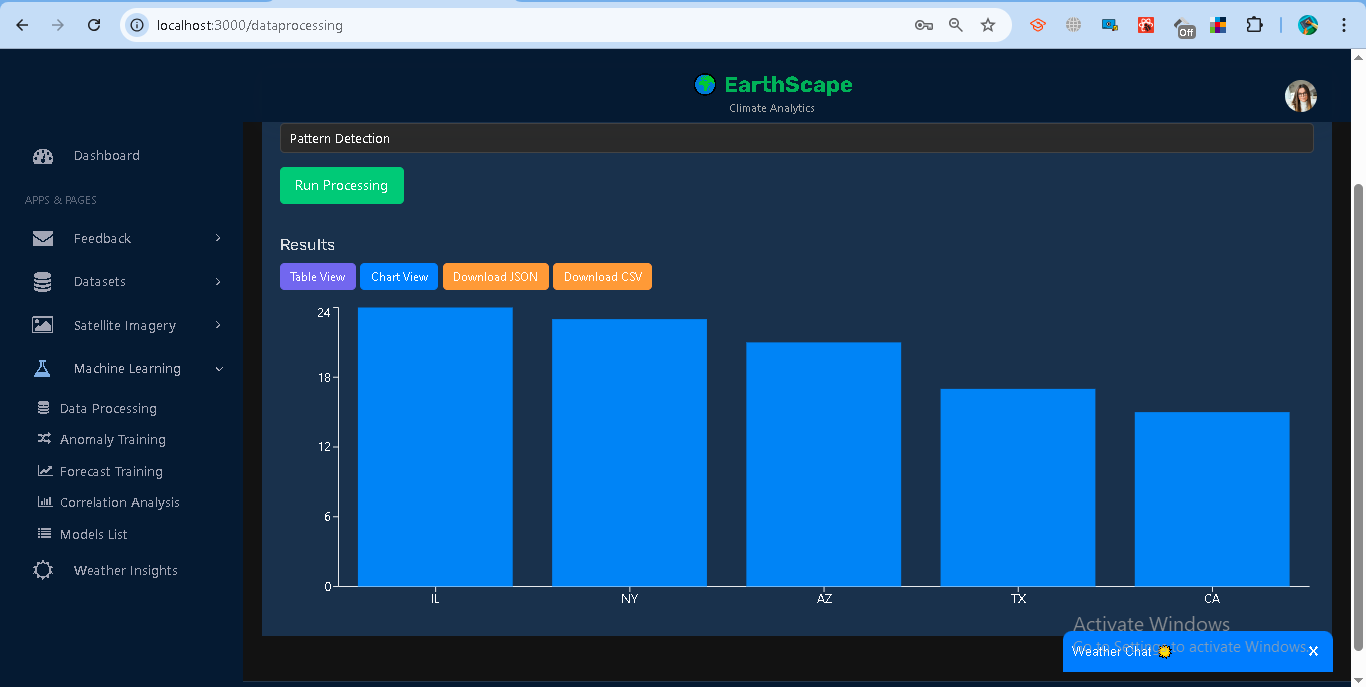
export default DataProcessing;

**How to Use (UI)**

1. Navigate to *Machine Learning → Data Processing*.
2. Select a dataset.
3. Choose an operation:  
   * For *average / group by / pattern*, select a field.
4. Click **Run Processing**.



1. View results as **Table** (with pagination) or **Chart** (Bar/Line).



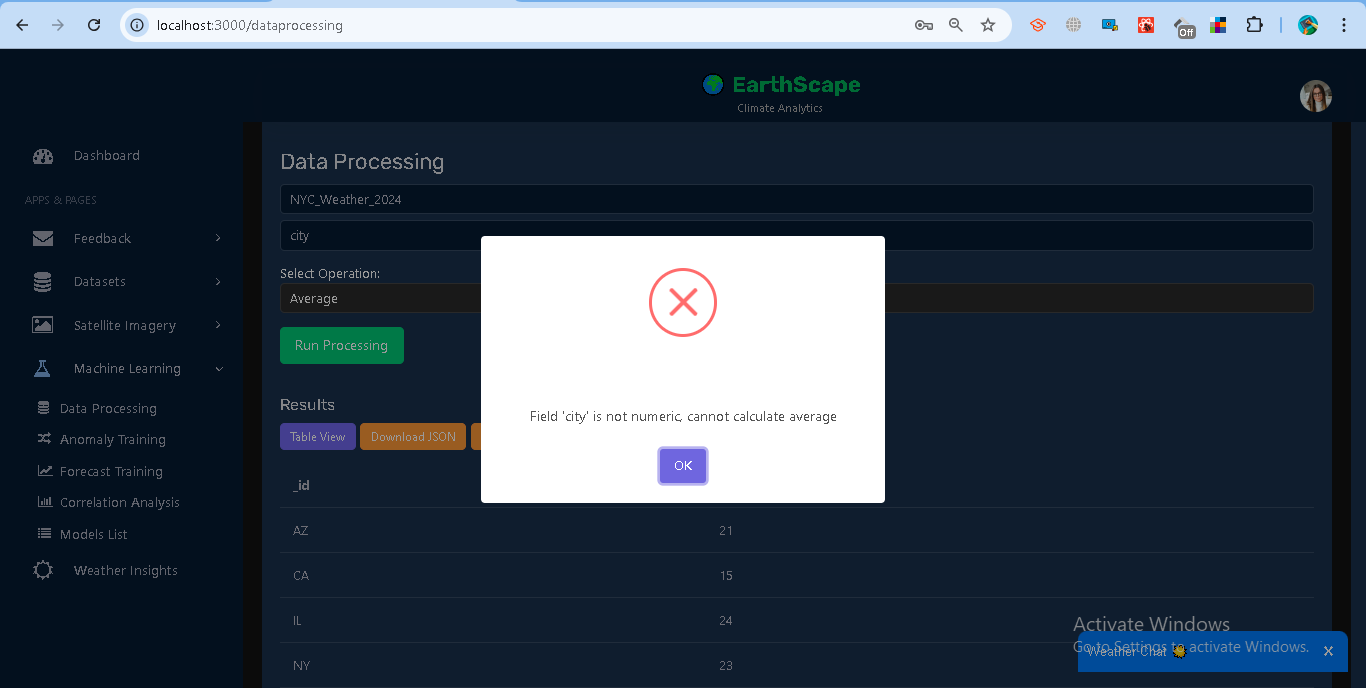
1. Download results as **JSON** or **CSV**.

**Examples**

* Count on NYC\_Weather\_2024 → { total: 1000 }
* Average on temperature → { average: 17.42 }
* Group By city → [{ \_id: "New York", count: 123 }, ...]
* Pattern on datetime → monthly counts { \_id: 1, count: 34 }

**Error Handling**

* Invalid field → "Field not numeric"
* Missing dataset → "Dataset not found"
* Expired/invalid token → "Unauthorized"



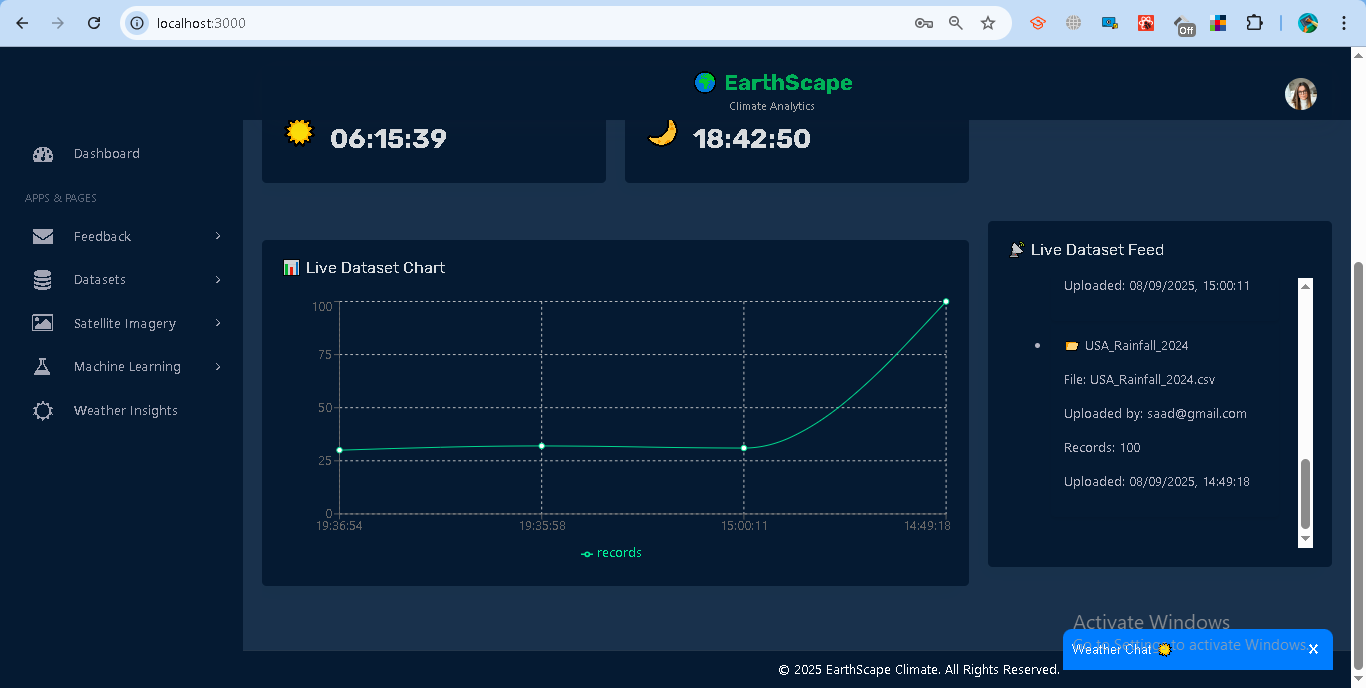
## **1.5 Real-time Data Processing**

The system integrates **real-time data streaming** with **batch data** for complete analysis. Users can view live updates on datasets and temperature records without refreshing the page.

### **Steps**

**Live Dataset Chart**

* Shows number of records over time.
* Loads historical data from /datasets/livefeed.
* Subscribes to /stream/datasets?token=... for new updates.



Code:  
# realtime.py

import os

import threading

import queue

import time

from collections import defaultdict

from flask import Blueprint, Response, request,jsonify, abort

from bson.json\_util import dumps

from bson.objectid import ObjectId

import jwt

from db import db

realtime\_bp = Blueprint("realtime\_bp", \_\_name\_\_)

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGO = os.getenv("JWT\_ALGORITHM", "HS256")

# ---- Subscribers per collection ----

\_subscribers = defaultdict(set)

\_subs\_lock = threading.Lock()

def \_add\_subscriber(collection\_name: str):

    q = queue.Queue(maxsize=1000)

    with \_subs\_lock:

        \_subscribers[collection\_name].add(q)

    return q

def \_remove\_subscriber(collection\_name: str, q: queue.Queue):

    with \_subs\_lock:

        \_subscribers[collection\_name].discard(q)

def \_broadcast(collection\_name: str, event: dict):

    with \_subs\_lock:

        dead = []

        for q in list(\_subscribers[collection\_name]):

            try:

                q.put\_nowait(event)

            except queue.Full:

                dead.append(q)

        for q in dead:

            \_subscribers[collection\_name].discard(q)

# ---- Watcher threads (one per collection) ----

\_watch\_threads\_started = {}

\_watch\_threads\_lock = threading.Lock()

def start\_watch\_thread(collection\_name: str):

    with \_watch\_threads\_lock:

        if \_watch\_threads\_started.get(collection\_name):

            return

        def watch():

            coll = db[collection\_name]

            pipeline = [

                {"$match": {"operationType": {

                    "$in": ["insert", "update", "replace"]}}}

            ]

            while True:

                try:

                    # Full document on updates so we can read user\_id, datasets, etc.

                    with coll.watch(pipeline=pipeline, full\_document="updateLookup") as stream:

                        for change in stream:

                            payload = {

                                "collection": collection\_name,

                                "op": change.get("operationType"),

                                "ts": time.time(),

                                "document": change.get("fullDocument"),

                                "documentKey": change.get("documentKey"),

                                "updateDescription": change.get("updateDescription"),

                            }

                            \_broadcast(collection\_name, payload)

                except Exception:

                    # Replica stepdown / transient errors → backoff then retry

                    time.sleep(1)

        t = threading.Thread(

            target=watch, name=f"watch-{collection\_name}", daemon=True)

        t.start()

        \_watch\_threads\_started[collection\_name] = True

# ---- Helper: find owner of globally-latest dataset (by embedded uploaded\_at) ----

def get\_owner\_of\_global\_latest\_dataset():

    """

    Returns {'user\_id': ObjectId, 'parent\_id': ObjectId} of the document

    that contains the globally latest dataset by datasets.uploaded\_at.

    """

    pipe = [

        {"$unwind": "$datasets"},

        {"$sort": {"datasets.uploaded\_at": -1}},

        {"$limit": 1},

        {"$project": {"user\_id": 1}}

    ]

    doc = next(db["datasets"].aggregate(pipe), None)

    if not doc:

        return None

    return {"user\_id": doc["user\_id"]}

# ---- SSE endpoint ----

@realtime\_bp.route("/stream/<collection\_name>")

def stream\_collection(collection\_name):

    """

    SSE stream with JWT in querystring: /stream/datasets?token=...

    Users: see only their own datasets' document changes.

    Admins: see only document changes for the owner of the globally-latest dataset.

    """

    token = request.args.get("token")

    if not token:

        return abort(401)

    # Decode token

    try:

        decoded = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGO])

        user\_id\_str = decoded.get("user\_id")

        # e.g., 'analyst' will be treated as user

        role = decoded.get("role", "user")

        if not user\_id\_str:

            return abort(401)

        user\_id = ObjectId(user\_id\_str)

    except Exception:

        return abort(401)

    # Determine which parent document(s) to allow

    # (Each user has one doc in 'datasets' with an embedded 'datasets' array)

    allowed\_user\_id = None

    if role == "admin":

        owner = get\_owner\_of\_global\_latest\_dataset()

        if not owner:

            return abort(404, "No dataset found")

        allowed\_user\_id = owner["user\_id"]

    else:

        # Any non-admin role (e.g., 'analyst', 'user', etc.)

        allowed\_user\_id = user\_id

    # Ensure a watcher is running

    start\_watch\_thread(collection\_name)

    q = \_add\_subscriber(collection\_name)

    def event\_stream():

        try:

            yield ": connected\n\n"

            while True:

                try:

                    event = q.get(timeout=20)

                    doc = event.get("document") or {}

                    doc\_user\_id = doc.get("user\_id")

                    # doc\_user\_id might be string or ObjectId depending on how it's stored

                    if isinstance(doc\_user\_id, str):

                        try:

                            doc\_user\_id = ObjectId(doc\_user\_id)

                        except Exception:

                            pass

                    # Filter: only forward events for the allowed user's document

                    if doc\_user\_id != allowed\_user\_id:

                        continue

               # 🔹 Detect if dataset array was updated

                    update\_desc = event.get("updateDescription", {})

                    updated\_fields = update\_desc.get("updatedFields", {})

                    new\_dataset = None

                    for field, value in updated\_fields.items():

                        if field.startswith("datasets.") and isinstance(value, dict):

                         new\_dataset = value  # this is the newly pushed dataset

                        break

                    if new\_dataset:

                         yield f"data: {dumps(new\_dataset)}\n\n"

                    else:

            # fallback small payload instead of whole doc

                      yield f"data: {dumps({'\_id': str(doc.get('\_id')), 'op': event.get('operationType')})}\n\n"

                except queue.Empty:

                    yield ": heartbeat\n\n"

        finally:

            \_remove\_subscriber(collection\_name, q)

    headers = {

        "Content-Type": "text/event-stream",

        "Cache-Control": "no-cache",

        "Connection": "keep-alive",

        "Access-Control-Allow-Origin": request.headers.get("Origin", "\*"),

    }

    return Response(event\_stream(), headers=headers)

@realtime\_bp.route("/stream/health")

def stream\_health():

    with \_subs\_lock, \_watch\_threads\_lock:

        return {

            "subscribers": {k: len(v) for k, v in \_subscribers.items()},

            "watchers": list(\_watch\_threads\_started.keys()),

            "db": str(db),

        }, 200

@realtime\_bp.route("/datasets/latest", methods=["GET"])

def get\_latest\_datasets():

    """

    Returns the most recent datasets (for the allowed user).

    """

    token = request.args.get("token")

    if not token:

        return abort(401)

    try:

        decoded = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGO])

        user\_id\_str = decoded.get("user\_id")

        role = decoded.get("role", "user")

        if not user\_id\_str:

            return abort(401)

        user\_id = ObjectId(user\_id\_str)

    except Exception:

        return abort(401)

    allowed\_user\_id = None

    if role == "admin":

        owner = get\_owner\_of\_global\_latest\_dataset()

        if not owner:

            return abort(404, "No dataset found")

        allowed\_user\_id = owner["user\_id"]

    else:

        allowed\_user\_id = user\_id

    doc = db["datasets"].find\_one({"user\_id": allowed\_user\_id})

    if not doc:

        return []

    return dumps(doc.get("datasets", [])), 200

@realtime\_bp.route("/datasets/livefeed", methods=["GET"])

def get\_datasets():

    # 1️⃣ Check Authorization header

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Authorization header missing"}), 401

    try:

        token = auth\_header.split(" ")[1]

        decoded = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGO])

        user\_id = decoded.get("user\_id")

        role = decoded.get("role")

        if not user\_id or not role:

            return jsonify({"error": "Invalid token payload"}), 401

    except jwt.ExpiredSignatureError:

        return jsonify({"error": "Token expired"}), 401

    except jwt.InvalidTokenError:

        return jsonify({"error": "Invalid token"}), 401

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    try:

        datasets\_collection = db.datasets

        # 2️⃣ Aggregation to fetch top 10 latest datasets

        match\_stage = {} if role == "admin" else {"user\_id": ObjectId(user\_id)}

        pipeline = [

            {"$match": match\_stage},

            {"$unwind": "$datasets"},

            {"$sort": {"datasets.uploaded\_at": -1}},

            {"$limit": 10},

            {

                "$project": {

                    "\_id": {"$toString": "$\_id"},

                    "user\_id": {"$toString": "$user\_id"},

                    "dataset\_name": "$datasets.dataset\_name",

                    "dataset\_type": "$datasets.dataset\_type",

                    "filename": "$datasets.filename",

                    "file\_path": "$datasets.file\_path",

                    "uploaded\_at": "$datasets.uploaded\_at",

                    "records\_count": {"$size": {"$ifNull": ["$datasets.records", []]}},

                    "uploaded\_by": "$datasets.uploaded\_by"

                }

            }

        ]

        latest\_datasets = list(datasets\_collection.aggregate(pipeline))

        # 3️⃣ Format dates

        for ds in latest\_datasets:

            ds["uploaded\_at"] = str(ds.get("uploaded\_at", "N/A"))

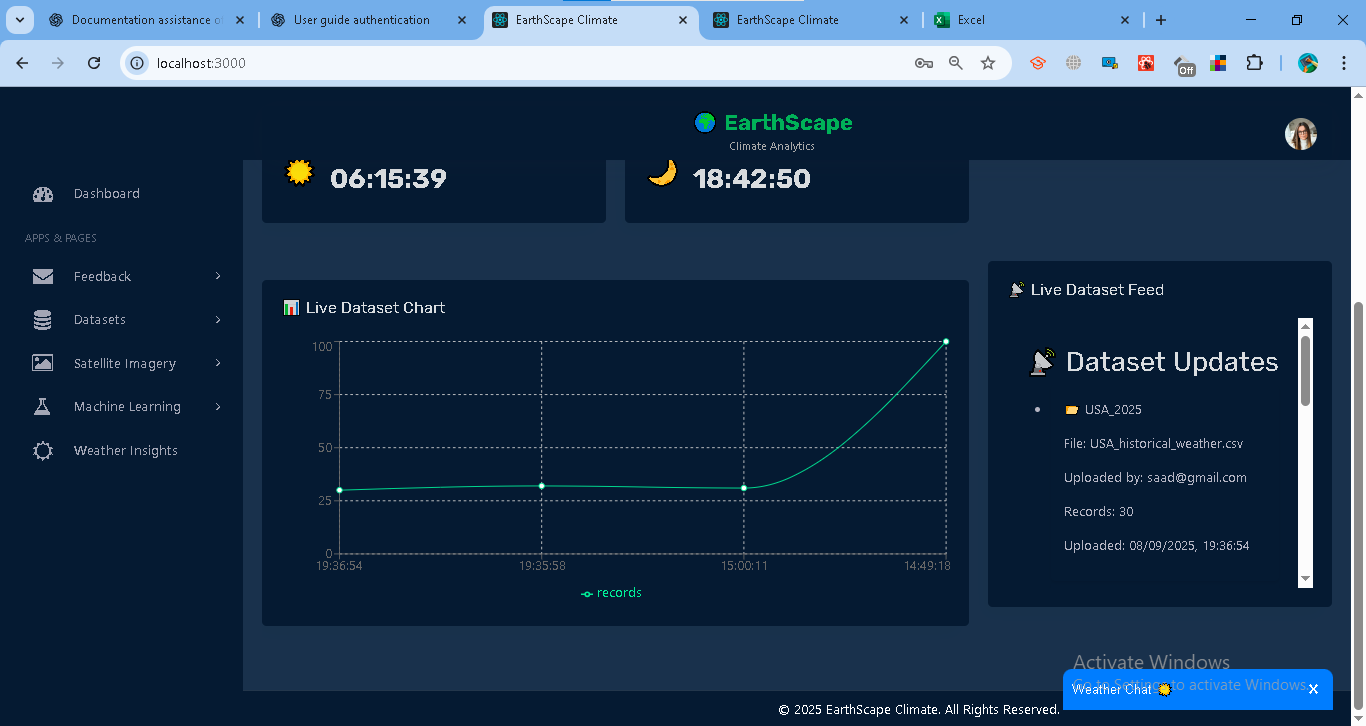
        return jsonify({"datasets": latest\_datasets}), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

**Live Dataset Feed**

* Lists the latest 10 datasets.
* Displays file, uploader, record count, and timestamp.
* Option to expand and view sample records.



**Live Temperature (per City)**

* Plots temperature trends across multiple cities.
* Loads recent records from /datasets/latest?token=....
* Streams updates via /stream/datasets?token=....
* Cities can be selected from a dropdown.



Code:

import React, { useState, useEffect } from "react";

import Select from "react-select";

import axios from "axios";

import { useEventSource } from "../../hooks/useEventSource";

import {

  LineChart,

  Line,

  XAxis,

  YAxis,

  Tooltip,

  CartesianGrid,

  Legend,

  ResponsiveContainer,

} from "recharts";

const LiveTemperatureCard = () => {

  const events = useEventSource("http://localhost:5000/api/stream/datasets");

  const [data, setData] = useState([]);

  const [cities, setCities] = useState([]);

  const [selectedCities, setSelectedCities] = useState([]);

  // ---- 1. Load bootstrap data once ----

  useEffect(() => {

    const fetchInitial = async () => {

      try {

        const token = localStorage.getItem("token");

const res = await axios.get(

  `http://localhost:5000/api/datasets/latest?token=${encodeURIComponent(token || "")}`

);

        const datasets = res.data ?? [];

        const records = datasets.flatMap((d) =>

          d.records.map((r) => ({

            timestamp: r.datetime,

            [r.city]: Number(r.temperature),

          }))

        );

        setData(records);

        const allCities = [

          ...new Set(records.flatMap((d) => Object.keys(d).filter((k) => k !== "timestamp"))),

        ];

        setCities(allCities);

        if (allCities.length > 0) {

          setSelectedCities(allCities.slice(0, 3)); // pick first 3

        }

      } catch (err) {

        console.error("Error fetching datasets:", err);

      }

    };

    fetchInitial();

  }, []);

  // ---- 2. Merge SSE events with existing ----

  useEffect(() => {

    if (events.length === 0) return;

    const newData = events.flatMap((e) => {

      const records = e.document?.datasets?.[0]?.records ?? [];

      return records.map((r) => ({

        timestamp: r.datetime,

        [r.city]: Number(r.temperature),

      }));

    });

    setData((prev) => {

      const merged = [...prev];

      newData.forEach((entry) => {

        const existing = merged.find((d) => d.timestamp === entry.timestamp);

        if (existing) {

          Object.assign(existing, entry);

        } else {

          merged.push(entry);

        }

      });

      return merged;

    });

    // also update cities if new appear

    const newCities = [

      ...new Set(newData.flatMap((d) => Object.keys(d).filter((k) => k !== "timestamp"))),

    ];

    if (newCities.length > 0) {

      setCities((prev) => Array.from(new Set([...prev, ...newCities])));

    }

  }, [events]);

  return (

    <div className="box">

      <div className="box-header no-border pb-0">

        <h4 className="box-title">🌡️ Real-Time Temperature (per City)</h4>

      </div>

      <div className="box-body">

        {/\* Multi-Select Dropdown \*/}

        <div className="mb-3">

          <label className="font-bold">Select Cities:</label>

          <Select

            isMulti

            options={cities.map((city) => ({ value: city, label: city }))}

            value={selectedCities.map((city) => ({ value: city, label: city }))}

            onChange={(selected) => setSelectedCities(selected.map((s) => s.value))}

            className="mt-2"

            placeholder="Choose cities..."

          />

        </div>

        {/\* Chart \*/}

        <ResponsiveContainer width="100%" height={350}>

          <LineChart data={data}>

            <CartesianGrid stroke="#ccc" />

            <XAxis dataKey="timestamp" />

            <YAxis />

            <Tooltip />

            <Legend />

            {selectedCities.map((city, idx) => (

              <Line

                key={city}

                type="monotone"

                dataKey={city}

                stroke={`hsl(${(idx \* 60) % 360}, 70%, 50%)`}

                dot={false}

              />

            ))}

          </LineChart>

        </ResponsiveContainer>

      </div>

    </div>

  );

};

export default LiveTemperatureCard;

### **Access Control**

* Analysts: see only their datasets.
* Admins: see datasets for the owner of the latest upload.

### **Error Handling**

* **401 Unauthorized** → Invalid or expired token.
* **404 No dataset found** → No datasets uploaded yet.
* **No updates** → Check SSE connection, token, and database stream settings.

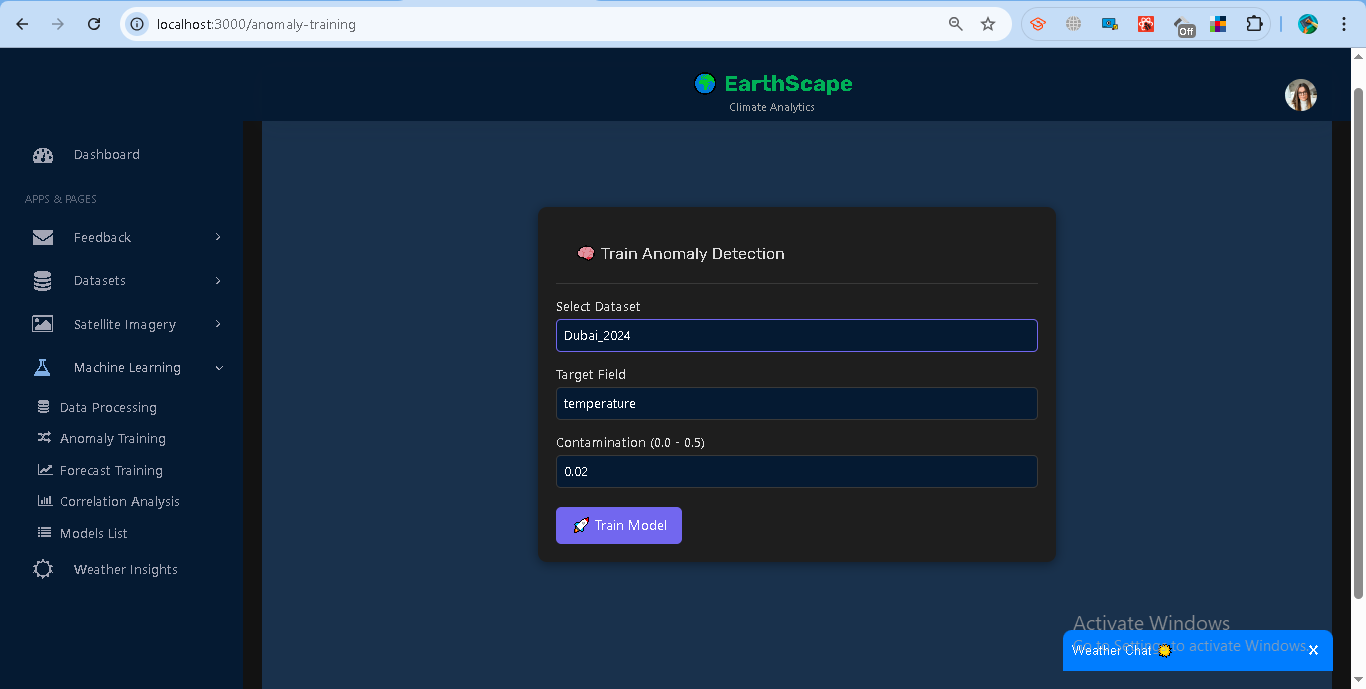
## **1.6 Anomaly Training**

The system provides anomaly detection using the **Isolation Forest model**. This helps identify unusual climate values (e.g., abnormal temperatures) in uploaded datasets.

### **Steps**

**Train a Model**

1. Go to **Machine Learning → Anomaly Training**.
2. Select a dataset (e.g., *Dubai\_2024*).
3. Enter the **Target Field** (numeric column, e.g., *temperature*).
4. Set **Contamination** (expected anomaly proportion, default = 0.02).
5. Click **Train Model**.



Code:

# ml\_blueprint.py

from flask import Blueprint, request, jsonify

from bson import ObjectId

import os, io, json

import numpy as np

import pandas as pd

from sklearn.ensemble import IsolationForest

import joblib

from statsmodels.tsa.arima.model import ARIMA

import warnings

import datetime as dt

warnings.filterwarnings("ignore")

from statsmodels.tsa.stattools import acf

# Reuse your existing app globals

from db import db

import jwt

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGO = os.getenv("JWT\_ALGORITHM", "HS256")

ml\_bp = Blueprint("ml\_bp", \_\_name\_\_)

MODELS\_DIR = os.path.join(os.getcwd(), "models")

os.makedirs(MODELS\_DIR, exist\_ok=True)

def \_auth\_ok():

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return (None, None, ("Authorization header missing", 401))

    try:

        token = auth\_header.split(" ")[1]

        decoded = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGO])

        user\_id = decoded.get("user\_id")

        role = decoded.get("role")

        if not user\_id or not role:

            return (None, None, ("Invalid token payload", 401))

        return (user\_id, role, None)

    except jwt.ExpiredSignatureError:

        return (None, None, ("Token expired", 401))

    except jwt.InvalidTokenError:

        return (None, None, ("Invalid token", 401))

    except Exception as e:

        return (None, None, (str(e), 401))

def \_guess\_time\_column(df):

    candidates = ["timestamp", "time", "datetime", "date", "uploaded\_at"]

    for c in candidates:

        if c in df.columns:

            return c

    return None

def \_load\_timeseries\_from\_mongo(user\_id, role, dataset\_name=None, dataset\_id=None, target\_field=None):

    """

    Loads records list from your db.datasets collection and returns a tidy DataFrame:

    columns: [ts, value] where ts is datetime64[ns], value is float

    """

    match\_stage = {} if role == "admin" else {"user\_id": ObjectId(user\_id)}

    pipe = [{"$match": match\_stage}, {"$unwind": "$datasets"}]

    if dataset\_id:

        pipe.append({"$match": {"datasets.\_id": ObjectId(dataset\_id)}})

    elif dataset\_name:

        pipe.append({"$match": {"datasets.dataset\_name": dataset\_name}})

    pipe.append({"$project": {

        "records": "$datasets.records",

        "uploaded\_at": "$datasets.uploaded\_at",

        "dataset\_name": "$datasets.dataset\_name"

    }})

    docs = list(db.datasets.aggregate(pipe))

    if not docs:

        raise ValueError("Dataset not found or no access")

    # Combine all records across matches; typical case is 1

    all\_records = []

    for d in docs:

        recs = d.get("records") or []

        # ensure it's a list of dicts

        if isinstance(recs, list):

            all\_records.extend(recs)

    if not all\_records:

        raise ValueError("No records found in the selected dataset")

    df = pd.DataFrame(all\_records)

    if df.empty:

        raise ValueError("Records are empty")

    if target\_field not in df.columns:

        raise ValueError(f"target\_field '{target\_field}' not present in dataset")

    # choose time column

    tcol = \_guess\_time\_column(df)

    if tcol is None:

        # fallback: use index order if no timestamp; still construct a pseudo time

        df["\_pseudo\_time"] = pd.date\_range(end=pd.Timestamp.utcnow(), periods=len(df), freq="min")

        tcol = "\_pseudo\_time"

    # coerce types

    df[tcol] = pd.to\_datetime(df[tcol], errors="coerce", utc=True)

    df = df.dropna(subset=[tcol])

    df = df.sort\_values(tcol)

    # numeric target

    df[target\_field] = pd.to\_numeric(df[target\_field], errors="coerce")

    df = df.dropna(subset=[target\_field])

    # return tidy

    return pd.DataFrame({"ts": df[tcol].values, "value": df[target\_field].values})

def \_make\_features(df, window=5):

    """

    Create simple, robust features for IsolationForest:

    value, 1-lag diff, rolling mean, rolling std (filled)

    """

    s = pd.Series(df["value"])

    lag1 = s.diff().fillna(0.0)

    roll\_mean = s.rolling(window, min\_periods=1).mean()

    roll\_std = s.rolling(window, min\_periods=1).std().fillna(0.0)

    X = np.column\_stack([s.values, lag1.values, roll\_mean.values, roll\_std.values])

    return X

def \_save\_model\_artifact(model, meta):

    model\_id = str(ObjectId())

    path = os.path.join(MODELS\_DIR, f"{model\_id}.pkl")

    joblib.dump({"model": model, "meta": meta}, path)

    meta\_doc = {

        "\_id": ObjectId(model\_id),

        "type": "anomaly\_isoforest",

        "created\_at": dt.datetime.utcnow(),

        "path": path,

        \*\*meta

    }

    db.models.insert\_one(meta\_doc)

    return model\_id

def \_load\_model\_artifact(model\_id, ensure\_type="anomaly\_isoforest"):

    doc = db.models.find\_one({"\_id": ObjectId(model\_id)})

    if not doc:

        raise ValueError("Model not found")

    if ensure\_type and doc.get("type") != ensure\_type:

        raise ValueError(f"Model type mismatch: expected {ensure\_type}, got {doc.get('type')}")

    blob = joblib.load(doc["path"])

    return blob["model"], blob["meta"], doc

@ml\_bp.route("/models", methods=["GET"])

def list\_models():

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    q = {} if role == "admin" else {"user\_id": ObjectId(user\_id)}

    cur = db.models.find(q).sort("created\_at", -1)

    out = []

    for d in cur:

        out.append({

            "model\_id": str(d["\_id"]),

            "type": d.get("type"),

            "dataset\_name": d.get("dataset\_name"),

            "target\_field": d.get("target\_field"),

            "created\_at": d.get("created\_at").isoformat() if d.get("created\_at") else None,

            "metrics": d.get("metrics"),

        })

    return jsonify({"models": out}), 200

@ml\_bp.route("/anomaly/train", methods=["POST"])

def anomaly\_train():

    """

    Train IsolationForest for anomaly detection.

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    payload = request.get\_json(force=True)

    dataset\_name = payload.get("dataset\_name")

    dataset\_id = payload.get("dataset\_id")

    target\_field = payload.get("target\_field")

    contamination = float(payload.get("contamination", 0.02))

    if not target\_field or (not dataset\_name and not dataset\_id):

        return jsonify({"error": "target\_field and (dataset\_name or dataset\_id) are required"}), 400

    try:

        tsdf = \_load\_timeseries\_from\_mongo(user\_id, role, dataset\_name, dataset\_id, target\_field)

        if len(tsdf) < 20:

            return jsonify({"error": "Not enough data to train (min 20 rows)"}), 400

        X = \_make\_features(tsdf)

        model = IsolationForest(n\_estimators=200, contamination=contamination, random\_state=42)

        model.fit(X)

        raw\_scores = model.decision\_function(X)

        preds = model.predict(X)

        anomaly\_rate = float((preds == -1).mean())

        meta = {

            "user\_id": ObjectId(user\_id),

            "role": role,

            "dataset\_name": dataset\_name,

            "dataset\_id": ObjectId(dataset\_id) if dataset\_id else None,

            "target\_field": target\_field,

            "contamination": contamination,

            "feature\_window": 5,

            "metrics": {

                "train\_rows": int(len(X)),

                "anomaly\_rate": anomaly\_rate,

                "score\_min": float(np.min(raw\_scores)),

                "score\_max": float(np.max(raw\_scores)),

                "score\_mean": float(np.mean(raw\_scores)),

            }

        }

        model\_id = \_save\_model\_artifact(model, meta)

        # --- ALERT CHECK + LOG ---

        triggered = []

        if len(tsdf) > 0:

            latest\_val = float(tsdf["value"].iloc[-1])

            latest\_ts = tsdf["ts"].iloc[-1]

            active\_alerts = list(db.alerts.find({

                "user\_id": ObjectId(user\_id),

                "dataset\_name": dataset\_name,

                "field": target\_field,

                "active": True

            }))

            for alert in active\_alerts:

                trig = None

                if alert["threshold\_type"] == "above" and latest\_val > alert["threshold\_value"]:

                    trig = f">{alert['threshold\_value']}"

                elif alert["threshold\_type"] == "below" and latest\_val < alert["threshold\_value"]:

                    trig = f"<{alert['threshold\_value']}"

                if trig:

                    log\_doc = {

                        "alert\_id": alert["\_id"],

                        "user\_id": ObjectId(user\_id),

                        "dataset\_name": dataset\_name,

                        "field": target\_field,

                        "value": latest\_val,

                        "ts": latest\_ts,

                        "triggered\_at": datetime.utcnow(),

                        "source": "anomaly\_train"

                    }

                    db.alert\_logs.insert\_one(log\_doc)

                    triggered.append({

                        "alert\_id": str(alert["\_id"]),

                        "rule": trig,

                        "value": latest\_val,

                        "ts": latest\_ts.isoformat()

                    })

        return jsonify({

            "model\_id": model\_id,

            "type": "anomaly\_isoforest",

            "metrics": meta["metrics"],

            "alerts\_triggered": triggered

        }), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

@ml\_bp.route("/anomaly/score", methods=["POST"])

def anomaly\_score():

    """

    Score dataset or new point(s) for anomalies.

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    payload = request.get\_json(force=True)

    model\_id = payload.get("model\_id")

    latest\_n = payload.get("latest\_n")

    new\_value = payload.get("value")

    if not model\_id:

        return jsonify({"error": "model\_id is required"}), 400

    try:

        model, meta, model\_doc = \_load\_model\_artifact(model\_id)

        if role != "admin" and str(model\_doc.get("user\_id")) != str(user\_id):

            return jsonify({"error": "Forbidden"}), 403

        tsdf = \_load\_timeseries\_from\_mongo(

            user\_id=user\_id if role != "admin" else str(model\_doc.get("user\_id")),

            role=role if role != "admin" else "admin",

            dataset\_name=meta.get("dataset\_name"),

            dataset\_id=str(meta.get("dataset\_id")) if meta.get("dataset\_id") else None,

            target\_field=meta["target\_field"]

        )

        if new\_value is not None:

            tsdf = pd.concat([

                tsdf,

                pd.DataFrame({"ts": [pd.Timestamp.utcnow()], "value": [float(new\_value)]})

            ], ignore\_index=True)

        if latest\_n:

            tsdf = tsdf.tail(int(latest\_n))

        X = \_make\_features(tsdf)

        scores = model.decision\_function(X)

        preds = model.predict(X)

        out = []

        for i, row in tsdf.reset\_index(drop=True).iterrows():

            out.append({

                "ts": pd.to\_datetime(row["ts"]).isoformat(),

                "value": float(row["value"]),

                "score": float(scores[i]),

                "is\_anomaly": bool(preds[i] == -1)

            })

        # --- ALERT CHECK + LOG ---

        triggered = []

        if len(out) > 0:

            latest\_val = out[-1]["value"]

            latest\_ts = out[-1]["ts"]

            active\_alerts = list(db.alerts.find({

                "user\_id": ObjectId(user\_id),

                "dataset\_name": meta["dataset\_name"],

                "field": meta["target\_field"],

                "active": True

            }))

            for alert in active\_alerts:

                trig = None

                if alert["threshold\_type"] == "above" and latest\_val > alert["threshold\_value"]:

                    trig = f">{alert['threshold\_value']}"

                elif alert["threshold\_type"] == "below" and latest\_val < alert["threshold\_value"]:

                    trig = f"<{alert['threshold\_value']}"

                if trig:

                    log\_doc = {

                        "alert\_id": alert["\_id"],

                        "user\_id": ObjectId(user\_id),

                        "dataset\_name": meta["dataset\_name"],

                        "field": meta["target\_field"],

                        "value": latest\_val,

                        "ts": latest\_ts,

                        "triggered\_at": datetime.utcnow(),

                        "source": "anomaly\_score"

                    }

                    db.alert\_logs.insert\_one(log\_doc)

                    triggered.append({

                        "alert\_id": str(alert["\_id"]),

                        "rule": trig,

                        "value": latest\_val,

                        "ts": latest\_ts

                    })

        return jsonify({

            "results": out[-(int(latest\_n) if latest\_n else len(out)):],

            "alerts\_triggered": triggered

        }), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

@ml\_bp.route("/models/<model\_id>", methods=["DELETE"])

def delete\_model(model\_id):

    user\_id, role, err = \_auth\_ok()

    if err:

        return jsonify({"error": err[0]}), err[1]

    try:

        # Find the model

        doc = db.models.find\_one({"\_id": ObjectId(model\_id)})

        if not doc:

            return jsonify({"error": "Model not found"}), 404

        # Permissions: only owner or admin

        if role != "admin" and str(doc.get("user\_id")) != str(user\_id):

            return jsonify({"error": "Forbidden"}), 403

        # Delete model file if exists

        path = doc.get("path")

        if path and os.path.exists(path):

            os.remove(path)

        # Delete from DB

        db.models.delete\_one({"\_id": ObjectId(model\_id)})

        return jsonify({"message": f"Model {model\_id} deleted successfully"}), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

# ========= STEP 2: FORECASTING =========

def \_infer\_freq(ts: pd.Series):

    # Try pandas to infer; fallback to median diff in minutes

    freq = pd.infer\_freq(ts)

    if freq is None and len(ts) >= 3:

        diffs = pd.Series(ts.sort\_values().diff().dropna().values).dt.total\_seconds()

        if not diffs.empty:

            med = diffs.median()

            # map common seconds to pandas freq strings

            mapping = {

                60: "T",           # 1 minute

                300: "5T",

                900: "15T",

                1800: "30T",

                3600: "H",

                86400: "D"

            }

            # find closest

            closest = min(mapping.keys(), key=lambda x: abs(x - med))

            freq = mapping[closest]

    return freq  # may still be None; ARIMA can handle but forecasting index will be range

def \_to\_timeseries(tsdf: pd.DataFrame):

    """Ensure a proper time-indexed series returned as (y, index, freq)."""

    s = pd.Series(tsdf["value"].astype(float).values, index=pd.to\_datetime(tsdf["ts"], utc=True))

    s = s.sort\_index()

    # 🚀 FIX: handle duplicate timestamps

    if s.index.has\_duplicates:

        # Option 1: keep first occurrence

        s = s[~s.index.duplicated(keep="first")]

        # Option 2: OR aggregate duplicates by mean:

        # s = s.groupby(s.index).mean()

    freq = \_infer\_freq(s.index.to\_series())

    if freq:

        s = s.asfreq(freq)

        s = s.interpolate(limit\_direction="both")

    return s, freq

@ml\_bp.route("/forecast/train", methods=["POST"])

def forecast\_train():

    """

    Train ARIMA on a dataset/time-series field.

    Body:

    {

      "dataset\_name": "karachi\_weather",  // or "dataset\_id": "..."

      "target\_field": "temperature",

      "p\_range": [0, 1, 2],               // optional small grid

      "d\_range": [0, 1, 2],               // optional

      "q\_range": [0, 1, 2]                // optional

    }

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    body = request.get\_json(force=True)

    dataset\_name = body.get("dataset\_name")

    dataset\_id = body.get("dataset\_id")

    target\_field = body.get("target\_field")

    p\_range = body.get("p\_range", [0, 1, 2])

    d\_range = body.get("d\_range", [0, 1, 2])

    q\_range = body.get("q\_range", [0, 1, 2])

    if not target\_field or (not dataset\_name and not dataset\_id):

        return jsonify({"error": "target\_field and (dataset\_name or dataset\_id) are required"}), 400

    try:

        tsdf = \_load\_timeseries\_from\_mongo(user\_id, role, dataset\_name, dataset\_id, target\_field)

        if len(tsdf) < 20:

            return jsonify({"error": "Not enough data to train (min 20 rows)"}), 400

        y, freq = \_to\_timeseries(tsdf)

        if y.isna().all():

            return jsonify({"error": "Series is all NaN after preprocessing"}), 400

        # Small grid search for (p,d,q) by AIC

        best = {"aic": float("inf"), "order": None, "model": None}

        for p in p\_range:

            for d in d\_range:

                for q in q\_range:

                    try:

                        m = ARIMA(y, order=(p, d, q))

                        res = m.fit()

                        if res.aic < best["aic"]:

                            best = {"aic": res.aic, "order": (p, d, q), "model": res}

                    except Exception:

                        continue

        if best["model"] is None:

            return jsonify({"error": "Failed to fit ARIMA on any (p,d,q) in the grid"}), 500

        res = best["model"]

        order = best["order"]

        meta = {

            "user\_id": ObjectId(user\_id),

            "role": role,

            "type": "forecast\_arima",

            "dataset\_name": dataset\_name,

            "dataset\_id": ObjectId(dataset\_id) if dataset\_id else None,

            "target\_field": target\_field,

            "order": {"p": order[0], "d": order[1], "q": order[2]},

            "freq": freq,

            "metrics": {

                "aic": float(best["aic"]),

                "train\_rows": int(len(y)),

            }

        }

        # Save fitted results (statsmodels results object)

        model\_id = \_save\_model\_artifact(res, meta)  # stores results + meta

        return jsonify({

            "model\_id": model\_id,

            "type": "forecast\_arima",

            "order": meta["order"],

            "freq": meta["freq"],

            "metrics": meta["metrics"]

        }), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

@ml\_bp.route("/forecast/predict", methods=["POST"])

def forecast\_predict():

    """

    Predict next horizon steps and auto-check + log alerts.

    Body:

    {

      "model\_id": "...",

      "horizon": 24

    }

    """

    user\_id, role, err = \_auth\_ok()

    if err:

        return jsonify({"error": err[0]}), err[1]

    body = request.get\_json(force=True)

    model\_id = body.get("model\_id")

    horizon = int(body.get("horizon", 24))

    if not model\_id:

        return jsonify({"error": "model\_id is required"}), 400

    try:

        model, meta, model\_doc = \_load\_model\_artifact(model\_id, ensure\_type="forecast\_arima")

        # permissions

        if role != "admin" and str(model\_doc.get("user\_id")) != str(user\_id):

            return jsonify({"error": "Forbidden"}), 403

        # Rebuild series

        tsdf = \_load\_timeseries\_from\_mongo(

            user\_id=user\_id if role != "admin" else str(model\_doc.get("user\_id")),

            role=role if role != "admin" else "admin",

            dataset\_name=meta.get("dataset\_name"),

            dataset\_id=str(meta.get("dataset\_id")) if meta.get("dataset\_id") else None,

            target\_field=meta["target\_field"]

        )

        y, freq = \_to\_timeseries(tsdf)

        # Refit ARIMA

        order = (meta["order"]["p"], meta["order"]["d"], meta["order"]["q"])

        refit = ARIMA(y, order=order).fit()

        forecast\_res = refit.get\_forecast(steps=horizon)

        mean = forecast\_res.predicted\_mean

        conf = forecast\_res.conf\_int(alpha=0.05)

        if freq:

            idx = pd.date\_range(

                start=y.index[-1] + pd.tseries.frequencies.to\_offset(freq),

                periods=horizon,

                freq=freq

            )

        else:

            idx = range(1, horizon + 1)

        out = []

        for i in range(horizon):

            ts\_val = (idx[i].isoformat() if hasattr(idx[i], "isoformat") else int(idx[i]))

            out.append({

                "ts": ts\_val,

                "yhat": float(mean.iloc[i]),

                "yhat\_lower": float(conf.iloc[i, 0]),

                "yhat\_upper": float(conf.iloc[i, 1])

            })

        # ---- ALERT CHECK + LOG ----

        triggered = []

        if len(out) > 0:

            latest\_point = out[-1]

            latest\_val = latest\_point["yhat"]

            latest\_ts = latest\_point["ts"]

            active\_alerts = list(db.alerts.find({

                "user\_id": ObjectId(user\_id),

                "dataset\_name": meta["dataset\_name"],

                "field": meta["target\_field"],

                "active": True

            }))

            for alert in active\_alerts:

                trig = None

                if alert["threshold\_type"] == "above" and latest\_val > alert["threshold\_value"]:

                    trig = f">{alert['threshold\_value']}"

                elif alert["threshold\_type"] == "below" and latest\_val < alert["threshold\_value"]:

                    trig = f"<{alert['threshold\_value']}"

                if trig:

                    log\_doc = {

                        "alert\_id": alert["\_id"],

                        "user\_id": ObjectId(user\_id),

                        "dataset\_name": meta["dataset\_name"],

                        "field": meta["target\_field"],

                        "value": latest\_val,

                        "ts": latest\_ts,

                        "triggered\_at": dt.datetime.utcnow(),

                        "source": "forecast\_predict"   # <--- added like anomaly\_train

                    }

                    db.alert\_logs.insert\_one(log\_doc)

                    triggered.append({

                        "alert\_id": str(alert["\_id"]),

                        "rule": trig,

                        "value": latest\_val,

                        "ts": latest\_ts

                    })

        return jsonify({

            "model\_id": model\_id,

            "target\_field": meta["target\_field"],

            "order": meta["order"],

            "freq": freq or meta.get("freq"),

            "forecast": out,

            "alerts\_triggered": triggered

        }), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

# STEP 3 Correlation Analysis (future)

@ml\_bp.route("/correlation/autocorr", methods=["POST"])

def correlation\_autocorr():

    """

    Compute autocorrelation (ACF) for a target field.

    Body:

    {

      "dataset\_name": "...",

      "target\_field": "...",

      "max\_lag": 50

    }

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    body = request.get\_json(force=True)

    dataset\_name = body.get("dataset\_name")

    target\_field = body.get("target\_field")

    max\_lag = int(body.get("max\_lag", 50))

    if not dataset\_name or not target\_field:

        return jsonify({"error": "dataset\_name and target\_field required"}), 400

    try:

        tsdf = \_load\_timeseries\_from\_mongo(user\_id, role, dataset\_name, target\_field=target\_field)

        y, freq = \_to\_timeseries(tsdf)

        acf\_vals = acf(y, nlags=max\_lag, fft=True)

        out = [{"lag": i, "corr": float(acf\_vals[i])} for i in range(len(acf\_vals))]

        return jsonify({"dataset\_name": dataset\_name, "target\_field": target\_field, "acf": out}), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

@ml\_bp.route("/correlation/cross", methods=["POST"])

def correlation\_cross():

    """

    Compute correlation between two dataset fields.

    Body:

    {

      "dataset1": {"name": "...", "field": "..."},

      "dataset2": {"name": "...", "field": "..."}

    }

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    body = request.get\_json(force=True)

    d1, f1 = body.get("dataset1", {}).get("name"), body.get("dataset1", {}).get("field")

    d2, f2 = body.get("dataset2", {}).get("name"), body.get("dataset2", {}).get("field")

    if not d1 or not f1 or not d2 or not f2:

        return jsonify({"error": "dataset1/dataset2 with name and field required"}), 400

    try:

        ts1 = \_load\_timeseries\_from\_mongo(user\_id, role, d1, target\_field=f1)

        ts2 = \_load\_timeseries\_from\_mongo(user\_id, role, d2, target\_field=f2)

        y1, \_ = \_to\_timeseries(ts1)

        y2, \_ = \_to\_timeseries(ts2)

        df = pd.concat([y1, y2], axis=1).dropna()

        corr = df.iloc[:,0].corr(df.iloc[:,1])

        return jsonify({

            "dataset1": {"name": d1, "field": f1},

            "dataset2": {"name": d2, "field": f2},

            "correlation": float(corr)

        }), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

@ml\_bp.route("/correlation/matrix", methods=["POST"])

def correlation\_matrix():

    """

    Compute correlation matrix across multiple dataset fields.

    Body:

    {

      "fields": [

        {"dataset": "...", "field": "..."},

        {"dataset": "...", "field": "..."}

      ]

    }

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    body = request.get\_json(force=True)

    fields = body.get("fields", [])

    if not fields or len(fields) < 2:

        return jsonify({"error": "at least 2 dataset fields required"}), 400

    try:

        dfs = []

        names = []

        for f in fields:

            ds, fld = f.get("dataset"), f.get("field")

            ts = \_load\_timeseries\_from\_mongo(user\_id, role, ds, target\_field=fld)

            y, \_ = \_to\_timeseries(ts)

            dfs.append(y.rename(f"{ds}.{fld}"))

            names.append(f"{ds}.{fld}")

        df\_all = pd.concat(dfs, axis=1).dropna()

        corr\_matrix = df\_all.corr().to\_dict()

        return jsonify({"matrix": corr\_matrix, "fields": names}), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

# --- ALERTS SYSTEM ---

@ml\_bp.route("/alerts/create", methods=["POST"])

def create\_alert():

    """

    Create a new alert rule.

    Body:

    {

      "dataset\_name": "karachi\_weather",

      "field": "temperature",

      "threshold\_type": "above",   # "above" | "below"

      "threshold\_value": 40

    }

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    body = request.get\_json(force=True)

    dataset\_name = body.get("dataset\_name")

    field = body.get("field")

    threshold\_type = body.get("threshold\_type")

    threshold\_value = body.get("threshold\_value")

    if not dataset\_name or not field or not threshold\_type or threshold\_value is None:

        return jsonify({"error": "Missing required fields"}), 400

    alert\_doc = {

        "user\_id": ObjectId(user\_id),

        "role": role,

        "dataset\_name": dataset\_name,

        "field": field,

        "threshold\_type": threshold\_type,

        "threshold\_value": float(threshold\_value),

        "active": True,

        "created\_at": datetime.utcnow()

    }

    db.alerts.insert\_one(alert\_doc)

    return jsonify({"message": "✅ Alert created", "alert": str(alert\_doc)}), 200

@ml\_bp.route("/alerts/fetch", methods=["GET"])

def fetch\_alerts():

    """

    Get all alerts for current user.

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    alerts = list(db.alerts.find({"user\_id": ObjectId(user\_id)}))

    for a in alerts:

        a["\_id"] = str(a["\_id"])

        a["user\_id"] = str(a["user\_id"])

    return jsonify({"alerts": alerts}), 200

@ml\_bp.route("/alerts/check", methods=["POST"])

def check\_alerts():

    """

    Check all active alerts for current user and return triggered ones.

    """

    user\_id, role, err = \_auth\_ok()

    if err: return jsonify({"error": err[0]}), err[1]

    active\_alerts = list(db.alerts.find({"user\_id": ObjectId(user\_id), "active": True}))

    triggered = []

    for alert in active\_alerts:

        try:

            tsdf = \_load\_timeseries\_from\_mongo(

                user\_id, role,

                dataset\_name=alert["dataset\_name"],

                dataset\_id=None,

                target\_field=alert["field"]

            )

            if len(tsdf) == 0:

                continue

            latest\_val = tsdf["value"].iloc[-1]

            if alert["threshold\_type"] == "above" and latest\_val > alert["threshold\_value"]:

                triggered.append({

                    "alert\_id": str(alert["\_id"]),

                    "dataset": alert["dataset\_name"],

                    "field": alert["field"],

                    "value": float(latest\_val),

                    "rule": f">{alert['threshold\_value']}"

                })

            elif alert["threshold\_type"] == "below" and latest\_val < alert["threshold\_value"]:

                triggered.append({

                    "alert\_id": str(alert["\_id"]),

                    "dataset": alert["dataset\_name"],

                    "field": alert["field"],

                    "value": float(latest\_val),

                    "rule": f"<{alert['threshold\_value']}"

                })

        except Exception as e:

            print("Error checking alert:", e)

    return jsonify({"triggered": triggered}), 200

@ml\_bp.route("/alerts/logs", methods=["GET"])

def get\_alert\_logs():

    """

    Fetch triggered alert logs for the authenticated user.

    Query params:

      ?limit=50     → limit number of logs (default 100)

      ?dataset\_name=...  → filter by dataset

      ?field=...         → filter by field

      ?alert\_id=...      → filter by a specific alert

    """

    user\_id, role, err = \_auth\_ok()

    if err:

        return jsonify({"error": err[0]}), err[1]

    try:

        limit = int(request.args.get("limit", 100))

        query = {}

        # Admins can view all logs, normal users only theirs

        if role != "admin":

            query["user\_id"] = ObjectId(user\_id)

        # Optional filters

        if "dataset\_name" in request.args:

            query["dataset\_name"] = request.args["dataset\_name"]

        if "field" in request.args:

            query["field"] = request.args["field"]

        if "alert\_id" in request.args:

            try:

                query["alert\_id"] = ObjectId(request.args["alert\_id"])

            except:

                return jsonify({"error": "Invalid alert\_id"}), 400

        logs = list(db.alert\_logs.find(query).sort("triggered\_at", -1).limit(limit))

        out = []

        for log in logs:

            out.append({

                "log\_id": str(log["\_id"]),

                "alert\_id": str(log["alert\_id"]),

                "dataset\_name": log["dataset\_name"],

                "field": log["field"],

                "value": log["value"],

                "ts": log["ts"].isoformat() if hasattr(log["ts"], "isoformat") else str(log["ts"]),

                "triggered\_at": log["triggered\_at"].isoformat() if hasattr(log["triggered\_at"], "isoformat") else str(log["triggered\_at"]),

                "source": log.get("source", "unknown")

            })

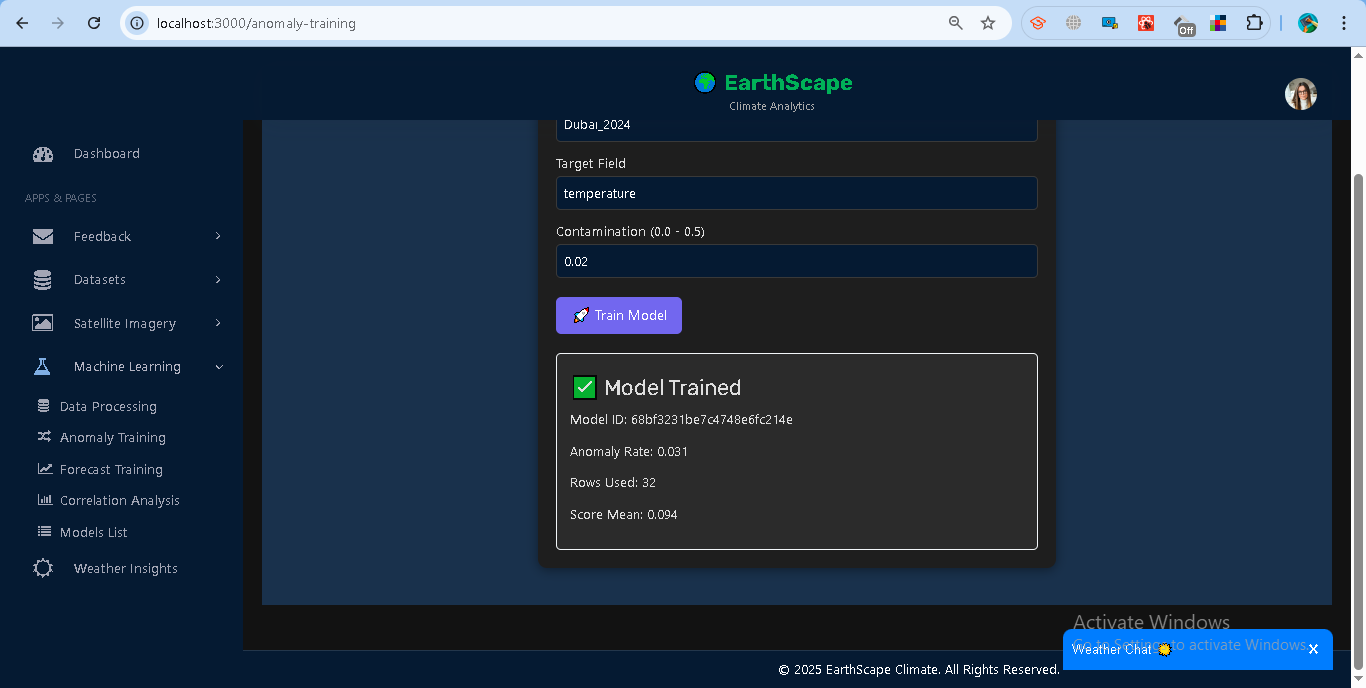
        return jsonify({"logs": out}), 200

    except Exception as e:

        return jsonify({"error": str(e)}), 500

**Results**

* Model ID is generated and stored.
* Training summary shows:  
  + **Anomaly Rate** → percentage of detected anomalies.
  + **Rows Used** → records used in training.
  + **Score Mean** → average decision score.

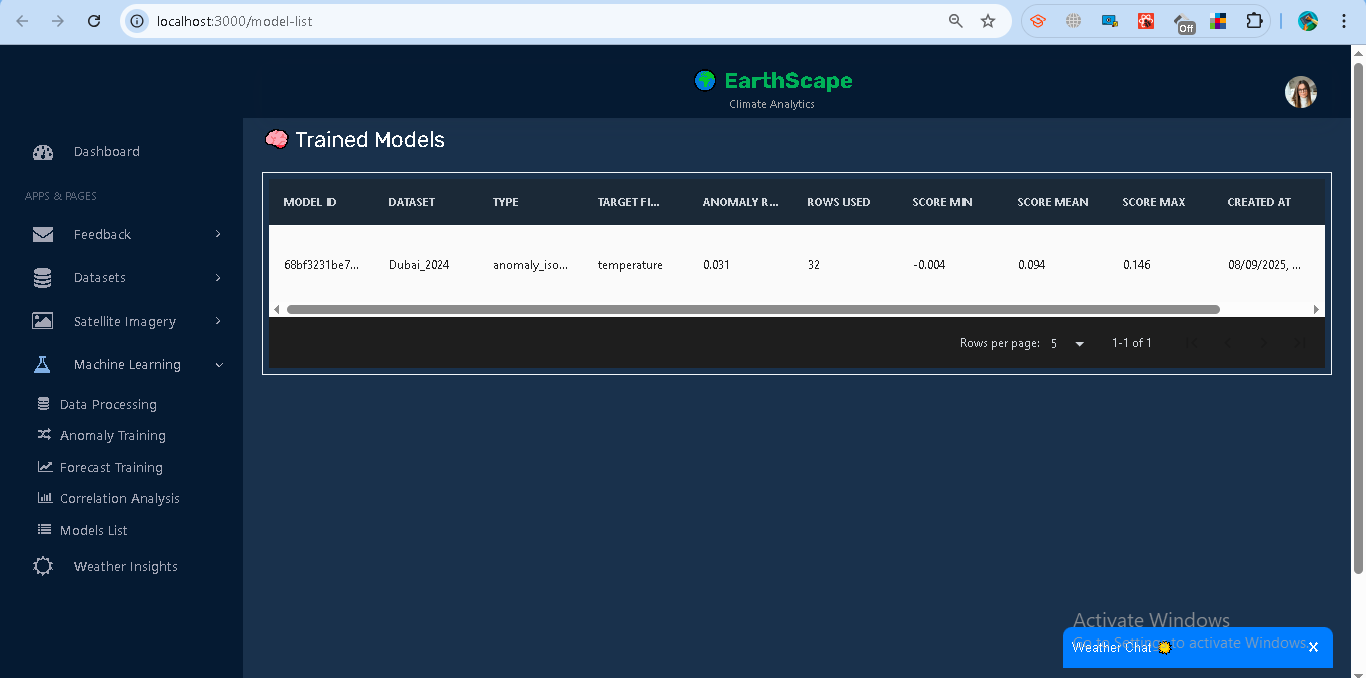


**Access Control**

* Analysts: Can only train and use models on their datasets.
* Admins: Can view and manage all users’ models.

**Error Handling**

* **400** → Missing dataset/field or insufficient records (<20).
* **401** → Invalid/expired token.
* **403** → Accessing another user’s model without admin rights.
* **500** → Training failure (invalid data format, missing numeric values).



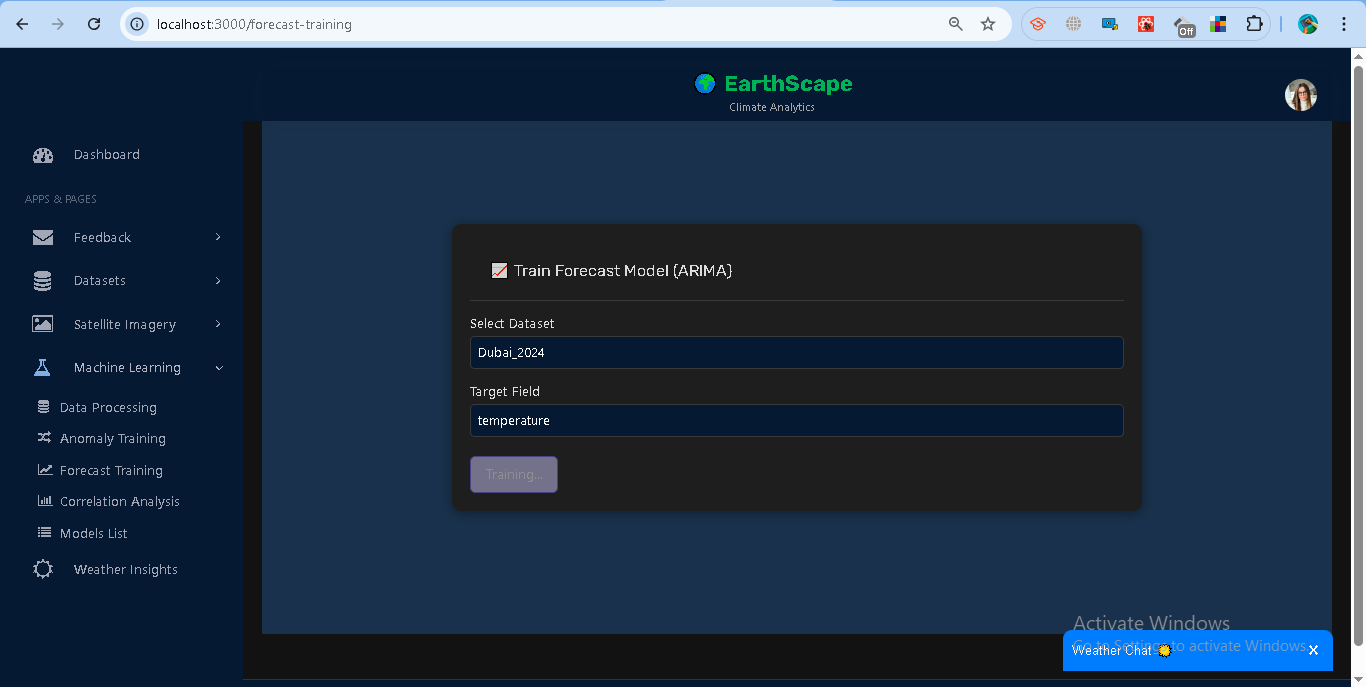
## **1.7 Forecast Training**

The system supports **time-series forecasting** using the **ARIMA model**. This allows predicting future values (e.g., temperature trends) from uploaded datasets.

### **Steps**

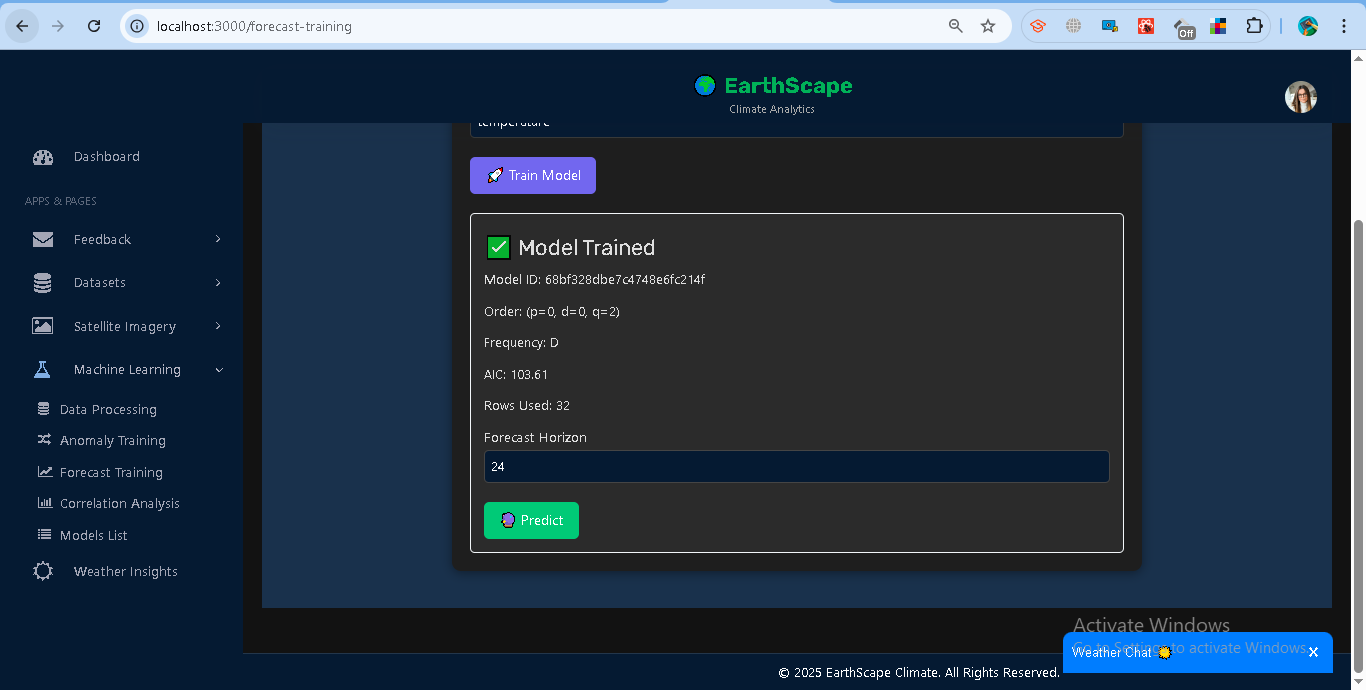
**Train a Forecast Model**

1. Go to **Machine Learning → Forecast Training**.
2. Select a dataset (e.g., *Dubai\_2024*).
3. Enter the **Target Field** (numeric column, e.g., *temperature*).
4. Click **Train Model**.



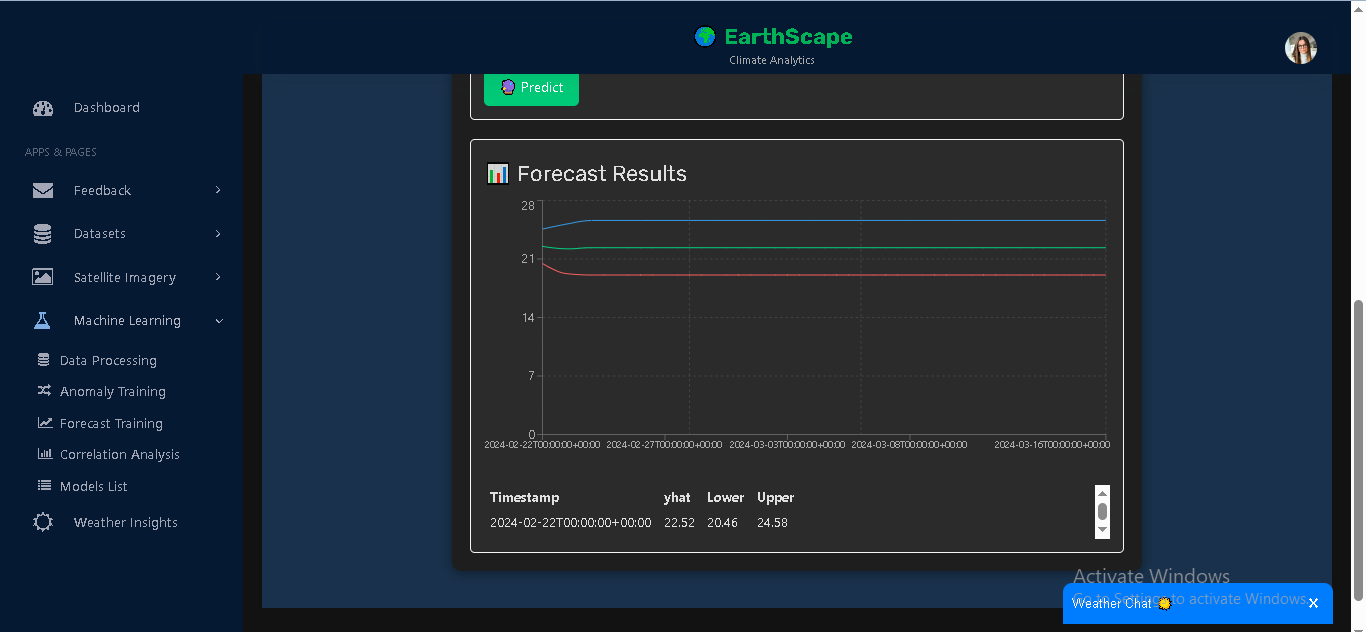
**Training Output**

* Model ID (saved in system).
* ARIMA Order (p,d,q).
* Frequency (detected time interval).
* Metrics: AIC score and training rows used.



**Generate Forecasts**

1. Enter a **Forecast Horizon** (number of future steps, e.g., 24).
2. Click **Predict**.
3. View results in:  
   * **Chart**: Predicted line with confidence bounds.
   * **Table**: Timestamp, prediction (yhat), lower and upper bounds.

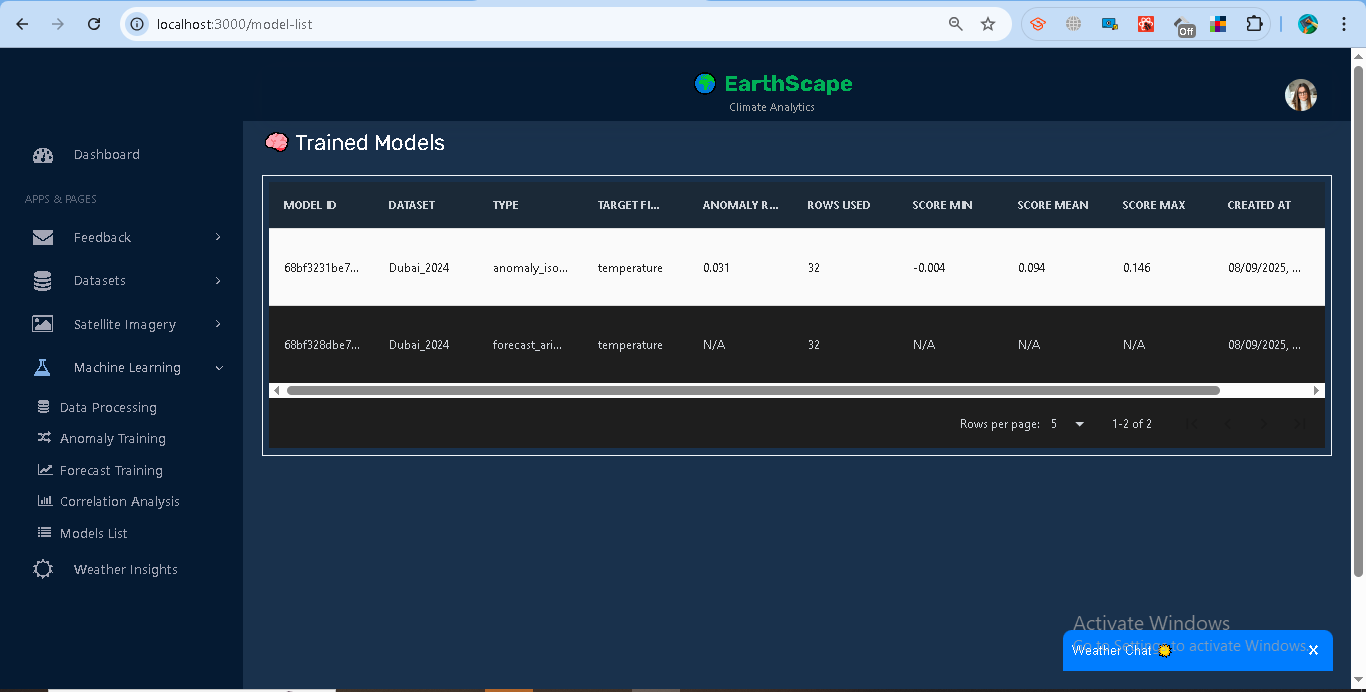


### **Access Control**

* Analysts: Train/predict only on their datasets.
* Admins: Can access and manage models across users.

### **Error Handling**

* **400** → Missing dataset/field or insufficient records (<20).
* **401** → Invalid/expired token.
* **403** → Not authorized to use another user’s model.
* **500** → Training/prediction error (data quality, ARIMA fit failure).



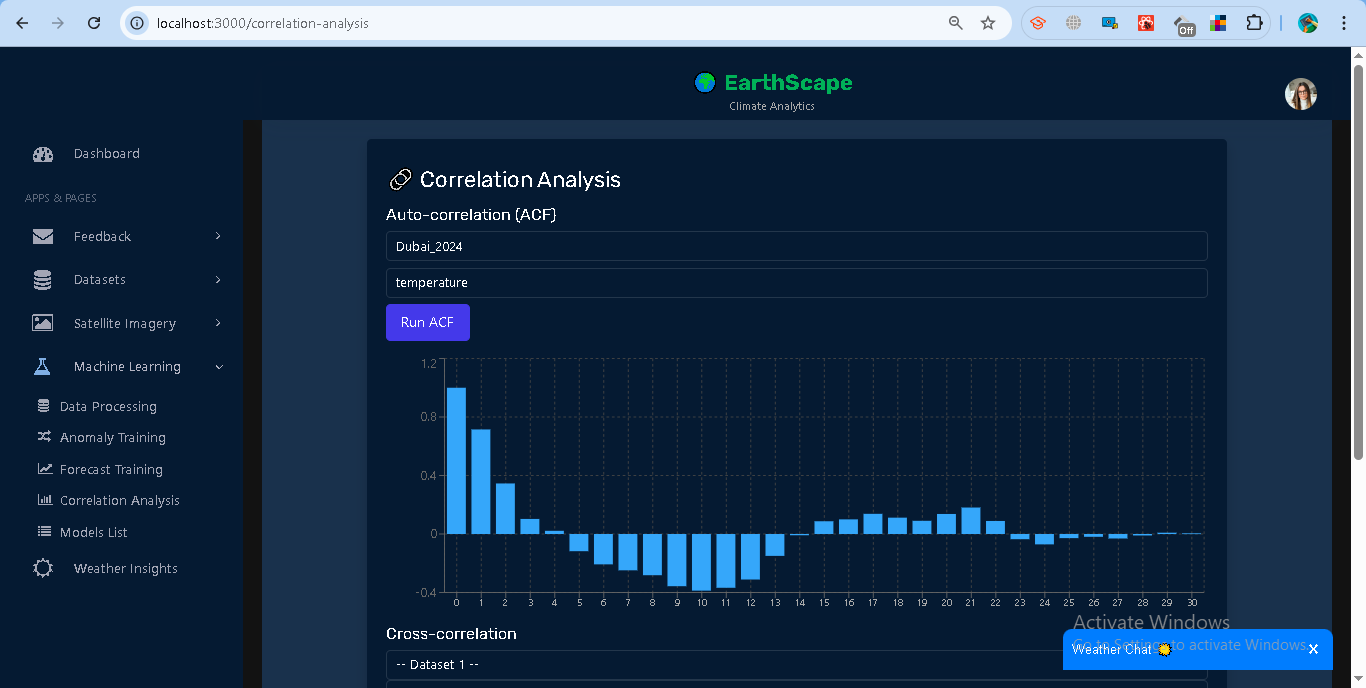
## **1.8 Correlation Analysis**

The system provides correlation tools to help understand relationships within and between climate datasets. Analysts can compute **Auto-correlation (ACF)**, **Cross-correlation**, and **Correlation Matrices**.

### **Features**

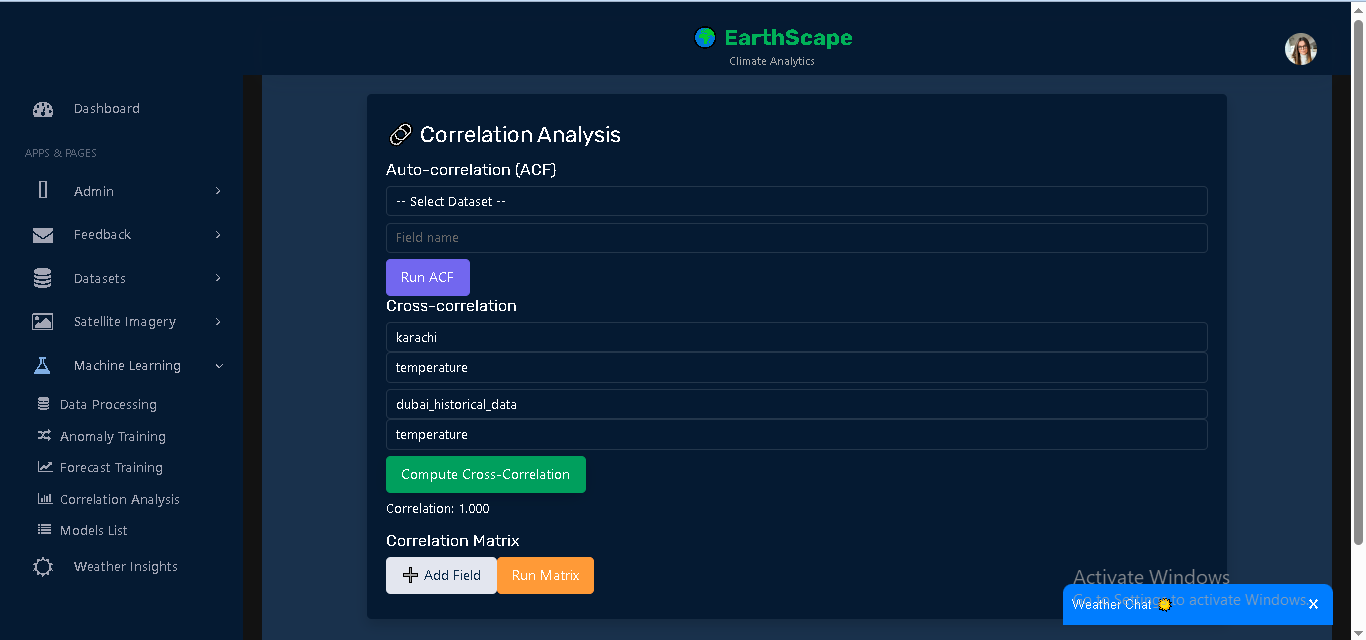
**Auto-correlation (ACF)**

* Measures how a field correlates with its past values (lags).
* Steps:  
  1. Select a dataset.
  2. Enter a **Target Field** (numeric column, e.g., *temperature*).
  3. Run ACF.
* Results: Bar chart of lag vs correlation.



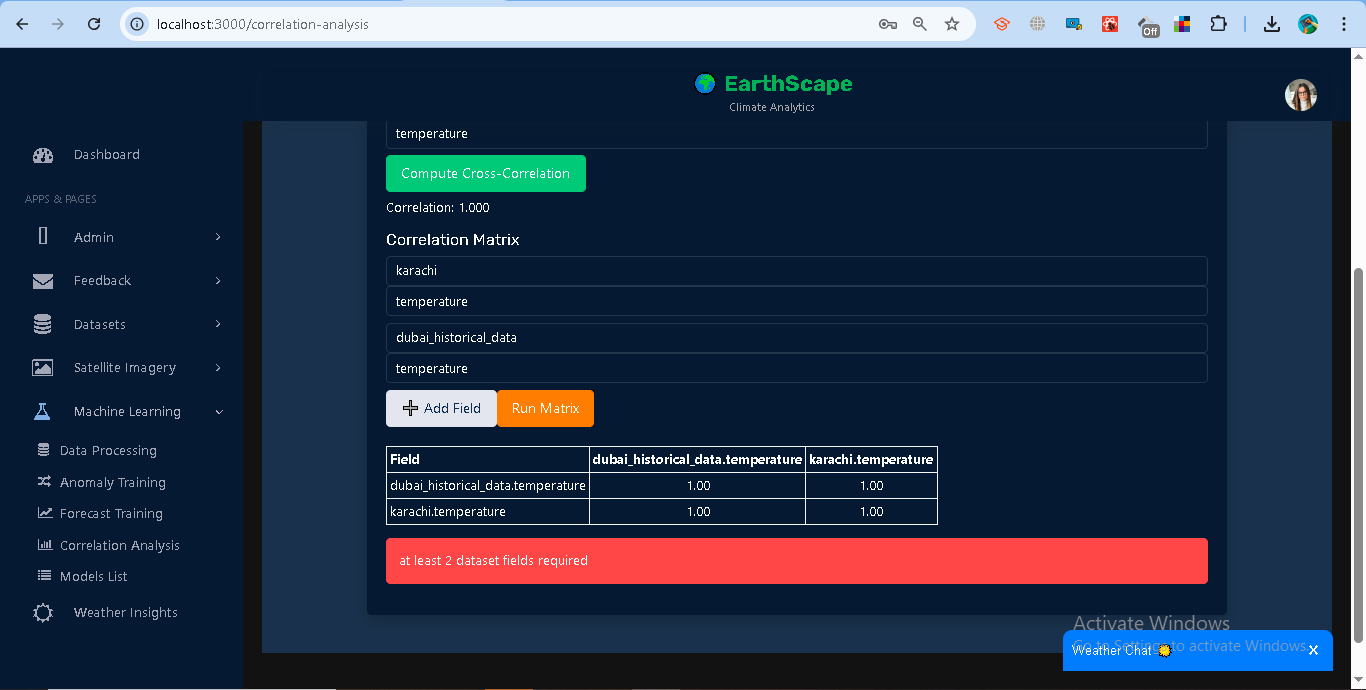
**Cross-correlation**

* Compares similarity between two dataset fields.
* Steps:  
  1. Select **Dataset 1 + Field**.
  2. Select **Dataset 2 + Field**.
  3. Compute correlation.
* Results: Single correlation score (between -1 and 1).



**Correlation Matrix**

* Shows correlation between multiple dataset fields at once.
* Steps:  
  1. Add datasets + fields (minimum 2).
  2. Run Matrix.
* Results: Table of pairwise correlation values.



### **Access Control**

* Analysts: Can analyze only their datasets.
* Admins: Can run correlations across all datasets.

### **Error Handling**

* **400** → Missing dataset/field inputs.
* **401** → Invalid/expired token.
* **403** → Accessing another user’s data without admin rights.
* **500** → Calculation error (e.g., missing numeric data).

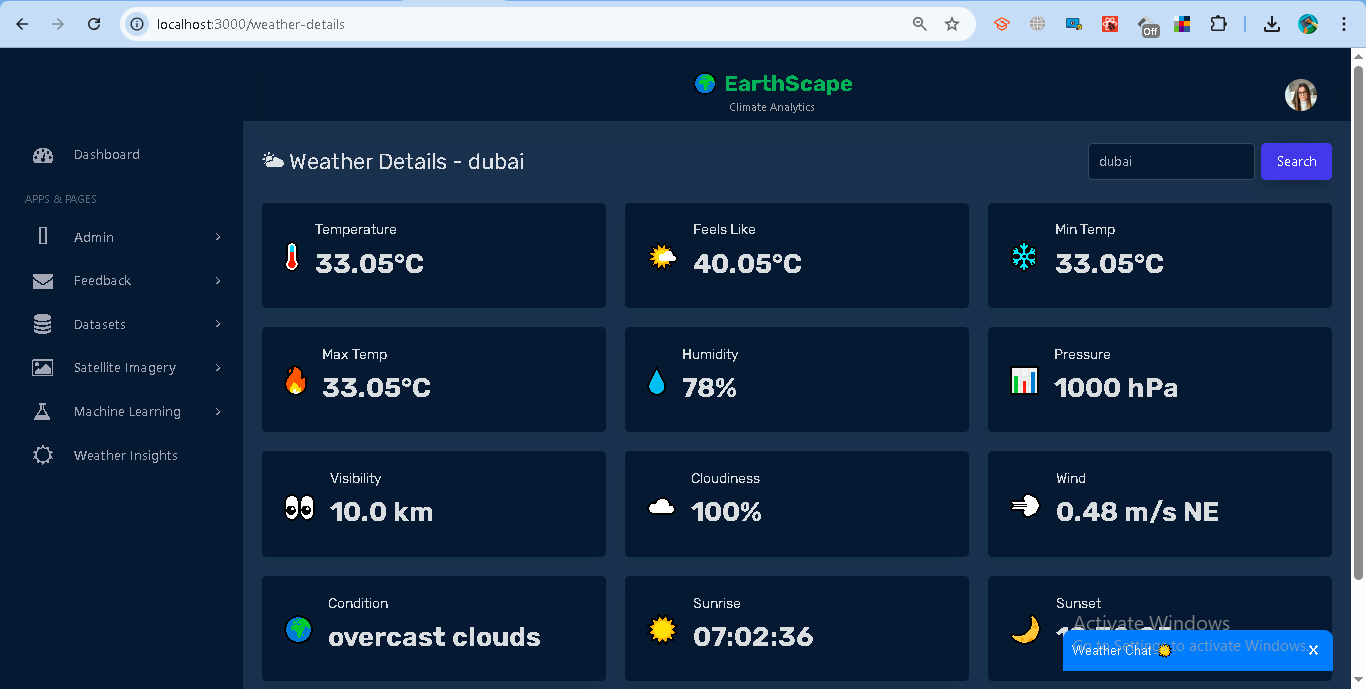
## **1.10 Weather Details**

The Weather Details module provides **real-time weather insights** using the OpenWeatherMap API. Users can search for a city and view key weather indicators in a dashboard-style card layout.

### **Features**

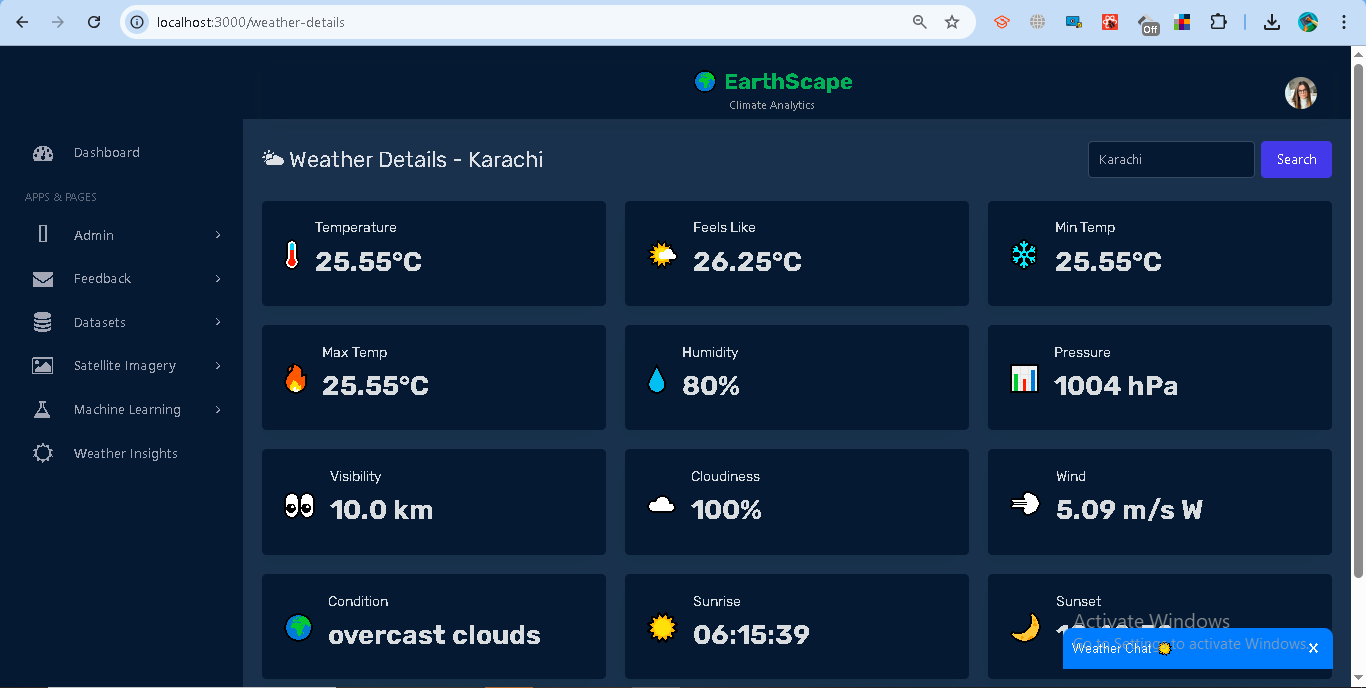
**Search and Fetch Weather**

1. Go to **Weather Insights** in the sidebar.
2. Enter a **city name** (e.g., *Dubai*) and click **Search**.
3. The system fetches live data from OpenWeatherMap.
4. The query is stored in MongoDB with a timestamp.



**Displayed Metrics**

* **Core:** Temperature, Feels Like, Min Temp, Max Temp.
* **Atmosphere:** Humidity, Pressure, Visibility, Cloudiness.
* **Wind & Condition:** Wind speed + direction, Weather condition.
* **Sun Info:** Sunrise and Sunset times (local).
* **Rain/Snow:** If available, rain/snow amount is displayed.



* **Errors:**
  + **400** → No city provided.
  + **404** → City not found.
  + **500** → External API/server error.

### **Access Control**

* Requires authentication (Bearer <token>).
* Searches are stored per user in MongoDB.

### **Error Handling**

* Toast notifications are shown for invalid city names or server errors.
* Weather card shows "Loading weather..." until data is available.

## **1.11 Weather ChatBot**

The Weather ChatBot provides an interactive way for users to query weather conditions for any city or country. It acts as a conversational assistant, fetching live data from the backend weather API.

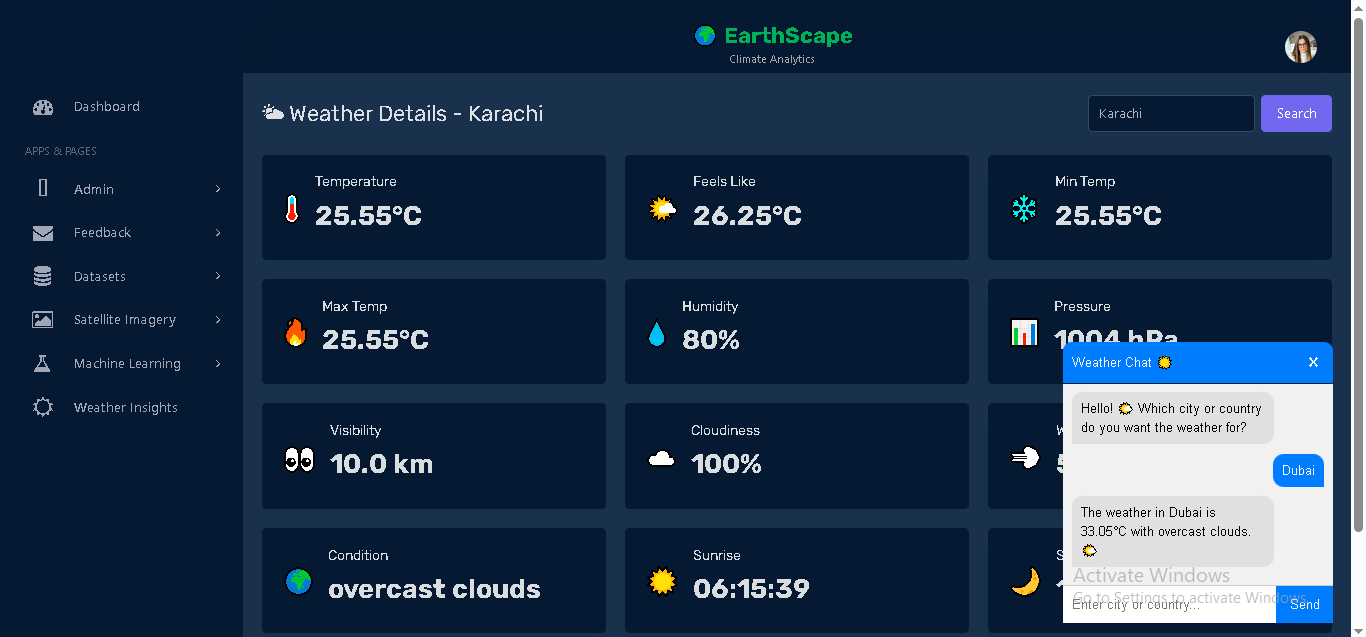
### **Features**

**Interactive Messaging**

* The bot greets the user and prompts for a city or country.
* Users type queries into the chat input.
* The bot responds with weather conditions such as **temperature** and **description**.

**Weather Query**

* Input: City or country name (e.g., *Dubai*, *USA*).
* Output: A conversational response, e.g.:  
   *“The weather in Dubai is 33°C with clear sky 🌤️”*

**

*Code:*from flask import Blueprint, request, jsonify

import requests

from db import search\_collection

import os

import datetime

weather\_bp = Blueprint("weather\_bp", \_\_name\_\_)

WEATHER\_API\_KEY = os.getenv("WEATHER\_API\_KEY")

@weather\_bp.route("/weather")

def get\_weather():

    city = request.args.get("city")

    if not city:

        return jsonify({"error": "Please provide a city or country name."}), 400

    try:

        # Capitalize first letter to help OpenWeatherMap

        city = city.strip().title()

        url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&units=metric&appid={WEATHER\_API\_KEY}"

        response = requests.get(url)

        response.raise\_for\_status()

        data = response.json()

        unix\_ts = data.get("dt")

        formatted\_date = datetime.datetime.utcfromtimestamp(unix\_ts).strftime("%Y/%m/%d") if unix\_ts else None

        # Save search to MongoDB

        search\_collection.insert\_one({

            "city": city,

            "timestamp": formatted\_date

        })

        result = {

            "city": data.get("name"),

            "temperature": data["main"]["temp"],

            "description": data["weather"][0]["description"],

            "icon": data["weather"][0]["icon"]

        }

        return jsonify(result)

    except requests.HTTPError:

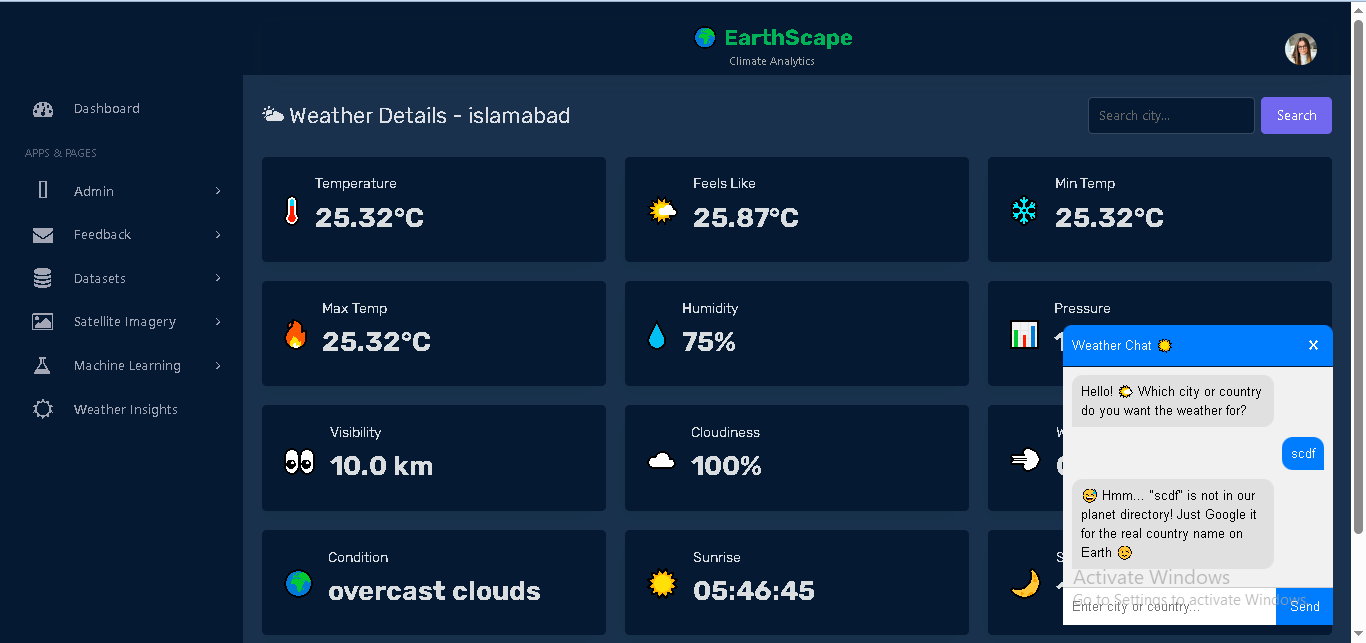
        return jsonify({"error": "City not found. Please enter a valid city or country."}), 404

    except Exception as e:

        return jsonify({"error": str(e)}), 500

**Error Handling**

* If an invalid city/country is entered, the bot replies with a playful error message:  
   *“😅 Hmm... ‘X’ is not in our planet directory! Just Google it for the real country name on Earth 😉”*

**

Code:

import React, { useState, useRef, useEffect } from "react";

import axios from "axios";

import "./ChatBot.css";

const ChatBot = () => {

  const [messages, setMessages] = useState([

    { sender: "bot", text: "Hello! 🌤️ Which city or country do you want the weather for?" }

  ]);

  const [input, setInput] = useState("");

  const [open, setOpen] = useState(false);

  const messagesEndRef = useRef(null);

  // Scroll to bottom whenever messages change

  const scrollToBottom = () => {

    messagesEndRef.current?.scrollIntoView({ behavior: "smooth" });

  };

  useEffect(() => {

    scrollToBottom();

  }, [messages]);

  const handleSend = async (e) => {

    e.preventDefault();

    if (!input) return;

    const userMessage = { sender: "user", text: input };

    setMessages((prev) => [...prev, userMessage]);

    try {

      const res = await axios.get(`http://127.0.0.1:5000/api/weather?city=${input}`);

      const data = res.data;

      if (data.error) {

        const botMessage = {

          sender: "bot",

          text: `😅 Hmm... "${input}" is not in our planet directory! Just Google it for the real country name on Earth 😉`

        };

        setMessages((prev) => [...prev, botMessage]);

      } else {

        const botMessage = {

          sender: "bot",

          text: `The weather in ${data.city} is ${data.temperature}°C with ${data.description}. 🌤️`

        };

        setMessages((prev) => [...prev, botMessage]);

      }

    } catch (err) {

      const botMessage = {

        sender: "bot",

        text: `😅 Hmm... "${input}" is not in our planet directory! Just Google it for the real country name on Earth 😉`

      };

      setMessages((prev) => [...prev, botMessage]);

    }

    setInput("");

  };

  return (

    <div className={`chatbot-container ${open ? "open" : ""}`}>

      <div className="chatbot-header" onClick={() => setOpen(!open)}>

        Weather Chat ☀️

        <button className="chatbot-close" onClick={() => setOpen(false)}>✖</button>

      </div>

      {open && (

        <div className="chatbot-body">

          <div className="chatbot-messages">

            {messages.map((msg, idx) => (

              <div key={idx} className={`chat-message ${msg.sender}`}>

                {msg.text}

              </div>

            ))}

            <div ref={messagesEndRef} />

          </div>

          <form className="chatbot-input" onSubmit={handleSend}>

            <input

              type="text"

              value={input}

              placeholder="Enter city or country..."

              onChange={(e) => setInput(e.target.value)}

            />

            <button type="submit">Send</button>

          </form>

        </div>

      )}

    </div>

  );

};

export default ChatBot;

**UI Behavior**

* The ChatBot is docked as a widget on the page.
* Clicking the header toggles the chatbot open/close.
* Messages auto-scroll to the bottom.
* Previous conversation is preserved while the session is active.

### **Access Control**

* Public feature, no login required.
* Requests go through the weather API with API key protection.

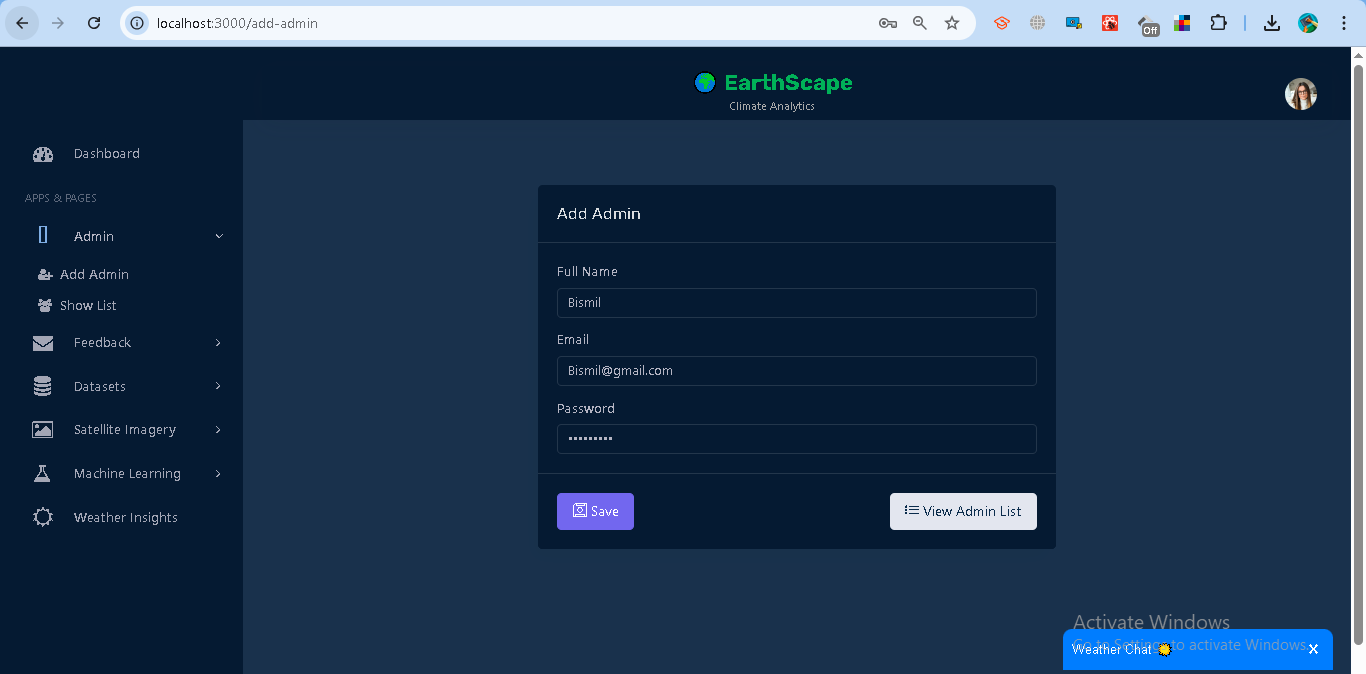
## **1.12 Admin Management**

The Admin Management module allows only users with the **admin role** to manage administrator accounts.

### **Features**

**Add Admin**

* Accessible only to admin users.
* Enter **Full Name, Email, Password**.
* On success, the new admin is stored in the system and a confirmation is displayed.



Code:

from flask import Blueprint, request, jsonify

from db import users\_collection

from passlib.hash import bcrypt

from datetime import datetime

import re, jwt, os

add\_admin\_bp = Blueprint("add\_admin\_bp", \_\_name\_\_)

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGORITHM = os.getenv("JWT\_ALGORITHM", "HS256")

@add\_admin\_bp.route("/add-admin", methods=["POST"])

def add\_admin():

    # Check token

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        if payload.get("role") != "admin":

            return jsonify({"error": "Only admins can add other admins"}), 403

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    data = request.get\_json()

    full\_name = data.get("full\_name", "").strip()

    email = data.get("email", "").strip().lower()

    password = data.get("password", "").strip()

    if not re.match(r"[^@]+@[^@]+\.[^@]+", email):

        return jsonify({"error": "Invalid email"}), 400

    if users\_collection.find\_one({"email": email}):

        return jsonify({"error": "Email already exists"}), 400

    hashed\_pw = bcrypt.hash(password)

    users\_collection.insert\_one({

        "full\_name": full\_name,

        "email": email,

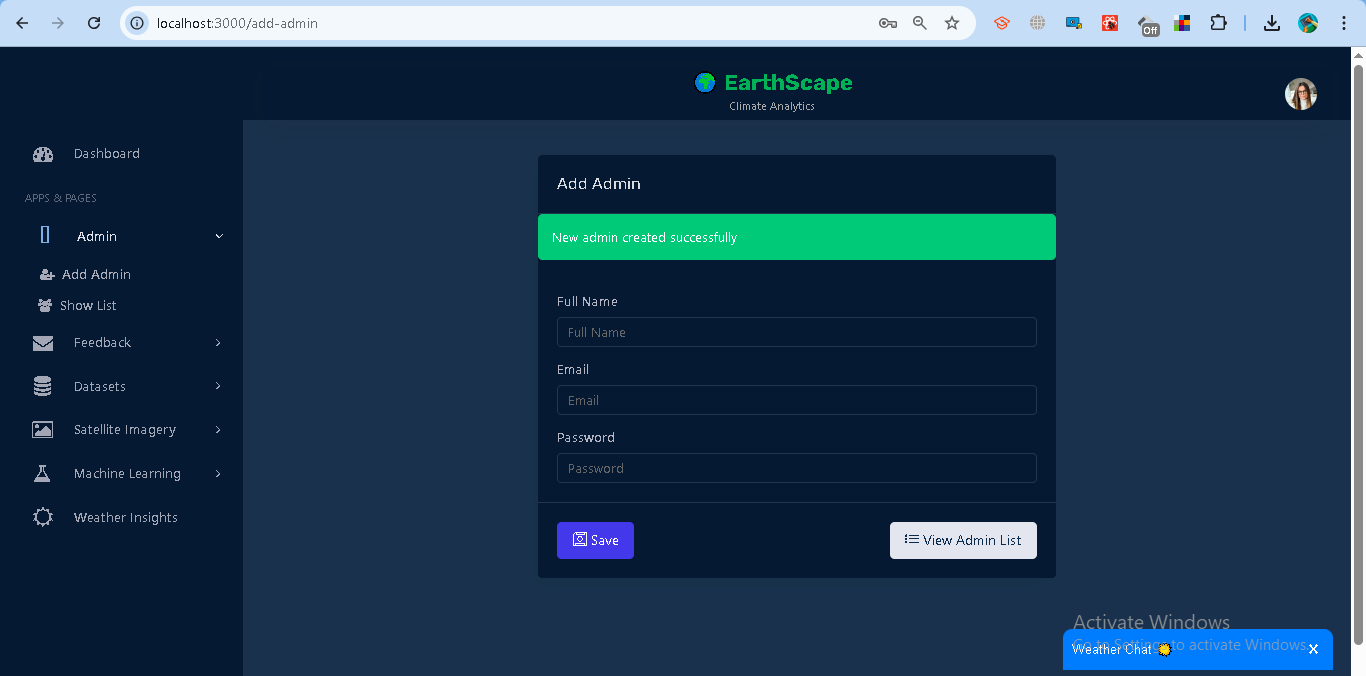
        "password": hashed\_pw,

        "role": "admin",

        "created\_at": datetime.utcnow()

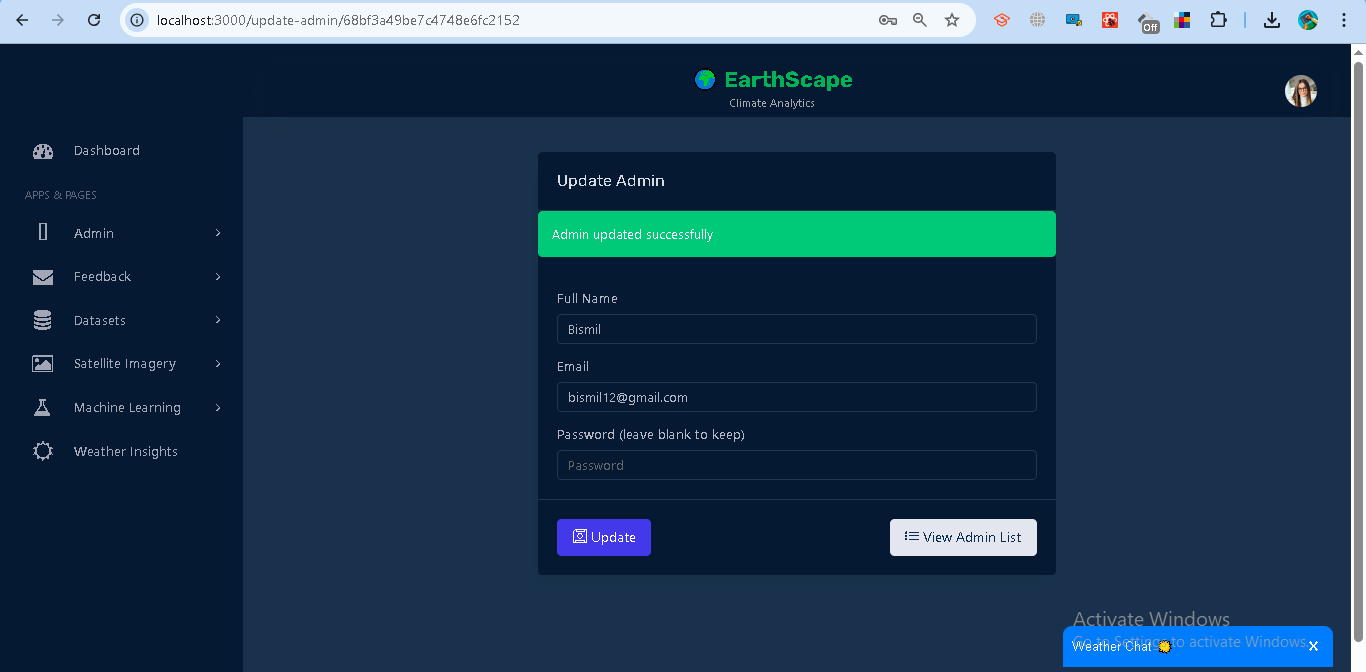
    })

    return jsonify({"message": "New admin created successfully"}), 201



**Update Admin**

* Admins can update existing admin details such as **Full Name, Email, Password**.
* Requires selecting an admin from the list and saving changes.



Code:

from flask import Blueprint, request, jsonify

from db import users\_collection

import jwt, os

from bson.objectid import ObjectId

from passlib.hash import bcrypt

from datetime import datetime

update\_admin\_bp = Blueprint("update\_admin\_bp", \_\_name\_\_)

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGORITHM = os.getenv("JWT\_ALGORITHM", "HS256")

@update\_admin\_bp.route("/update-admin/<admin\_id>", methods=["PUT"])

def update\_admin(admin\_id):

    # Check token

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        if payload.get("role") != "admin":

            return jsonify({"error": "Only admins can update other admins"}), 403

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    data = request.get\_json()

    full\_name = data.get("full\_name", "").strip()

    email = data.get("email", "").strip().lower()

    password = data.get("password", "").strip()

    update\_data = {}

    if full\_name:

        update\_data["full\_name"] = full\_name

    if email:

        update\_data["email"] = email

    if password:

        update\_data["password"] = bcrypt.hash(password)

    if not update\_data:

        return jsonify({"error": "No data to update"}), 400

    try:

        result = users\_collection.update\_one(

            {"\_id": ObjectId(admin\_id), "role": "admin"},

            {"$set": update\_data}

        )

        if result.matched\_count == 0:

            return jsonify({"error": "Admin not found"}), 404

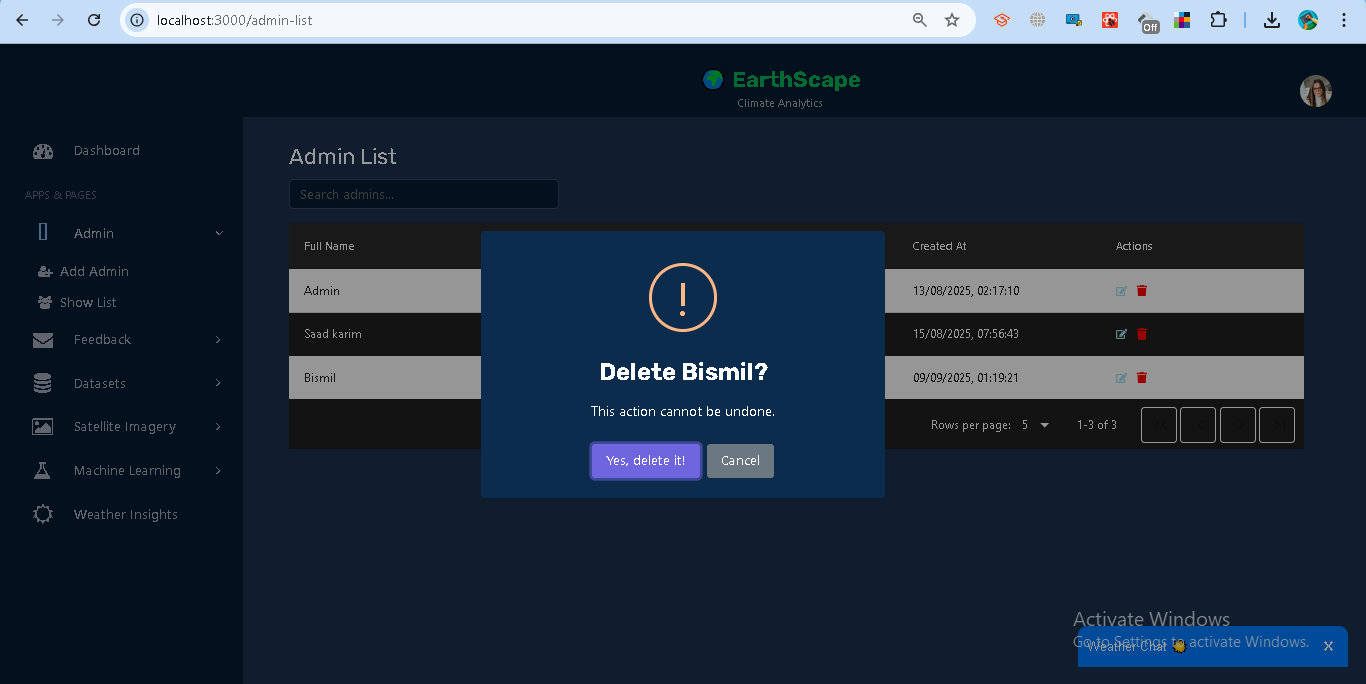
    except Exception as e:

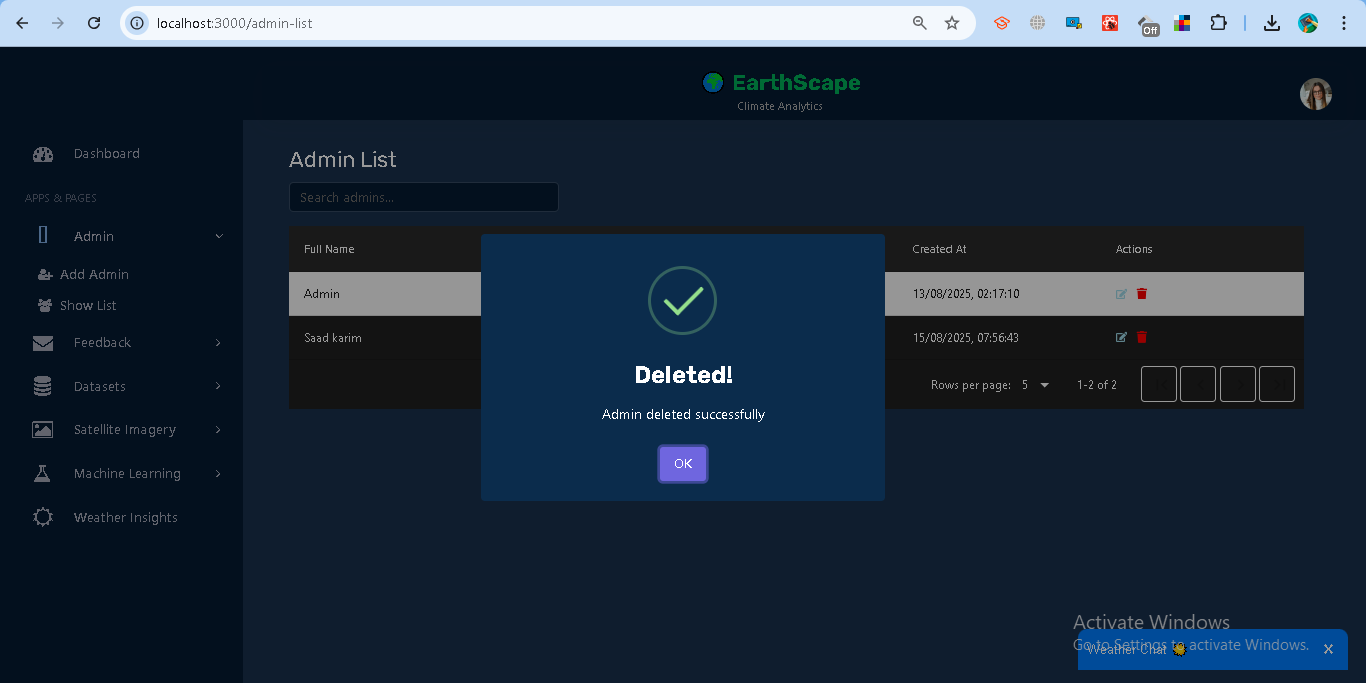
        return jsonify({"error": "Invalid admin ID"}), 400

    return jsonify({"message": "Admin updated successfully"}), 200

**Delete Admin**

* Admins can remove an existing admin from the system.
* Requires confirmation before deletion.



  
Code:

from flask import Blueprint, request, jsonify

from db import users\_collection

import jwt, os

from bson.objectid import ObjectId

delete\_admin\_bp = Blueprint("delete\_admin\_bp", \_\_name\_\_)

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGORITHM = os.getenv("JWT\_ALGORITHM", "HS256")

@delete\_admin\_bp.route("/delete-admin/<admin\_id>", methods=["DELETE"])

def delete\_admin(admin\_id):

    # Check token

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        if payload.get("role") != "admin":

            return jsonify({"error": "Only admins can delete other admins"}), 403

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    try:

        # Convert string \_id to ObjectId

        result = users\_collection.delete\_one({"\_id": ObjectId(admin\_id), "role": "admin"})

        if result.deleted\_count == 0:

            return jsonify({"error": "Admin not found or cannot delete non-admin"}), 404

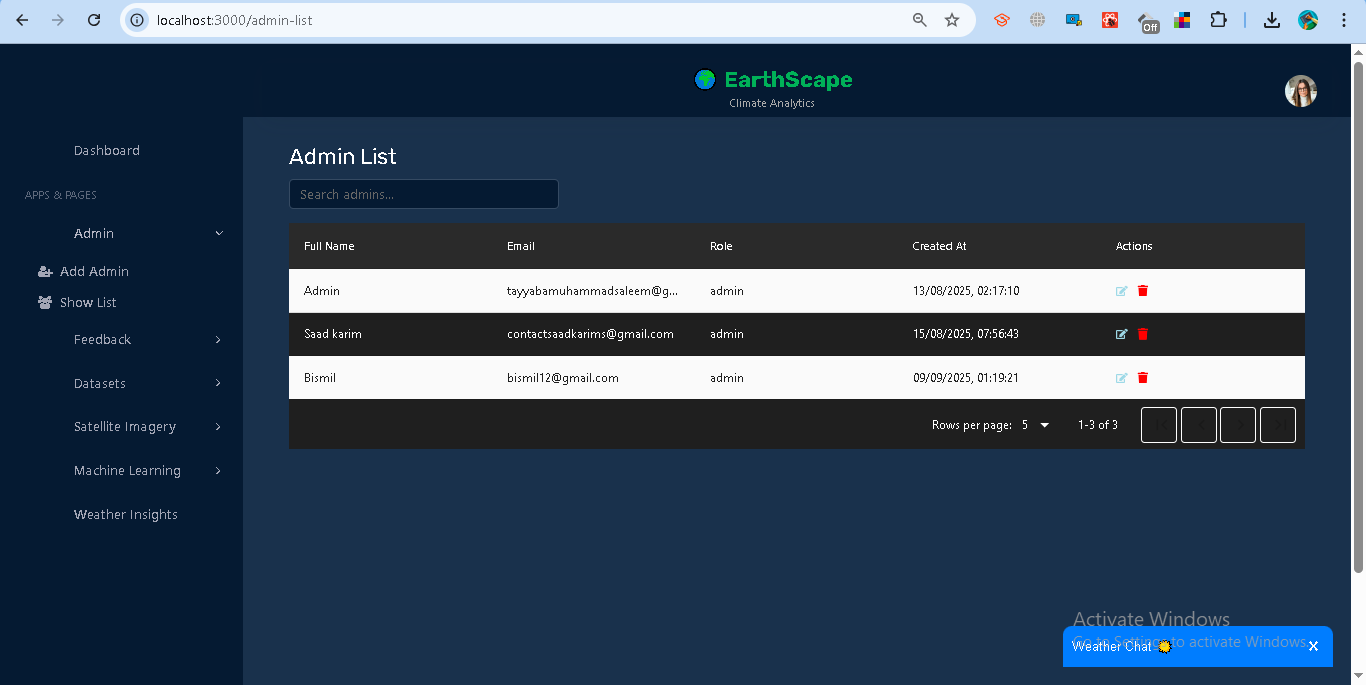
    except Exception as e:

        return jsonify({"error": "Invalid admin ID"}), 400

    return jsonify({"message": "Admin deleted successfully"}), 200

**View Admin List**

* Displays all registered admin users.
* Provides options to update or delete each admin.



Code:

from flask import Blueprint, jsonify, request

from db import users\_collection

import jwt, os

admin\_list\_bp = Blueprint("admin\_list\_bp", \_\_name\_\_)

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGORITHM = os.getenv("JWT\_ALGORITHM", "HS256")

@admin\_list\_bp.route("/admin-list", methods=["GET"])

def get\_admin\_list():

    # Check token

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        if payload.get("role") != "admin":

            return jsonify({"error": "Only admins can view the list"}), 403

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    # Fetch only admins

    admins = list(users\_collection.find({"role": "admin"}))

    # Convert ObjectId to string for React usage

    for admin in admins:

        admin["\_id"] = str(admin["\_id"])

    return jsonify(admins), 200

### **Access Control**

* Only users with role = **admin** can access this module.
* Non-admins are redirected and shown an **Access Denied** warning.

### **Error Handling**

* **Missing Fields** → Prompts user to fill all required inputs.
* **Unauthorized** → User must log in to access.
* **Forbidden** → Only admins can perform this action.
* **Server Errors** → Displayed with appropriate error messages.

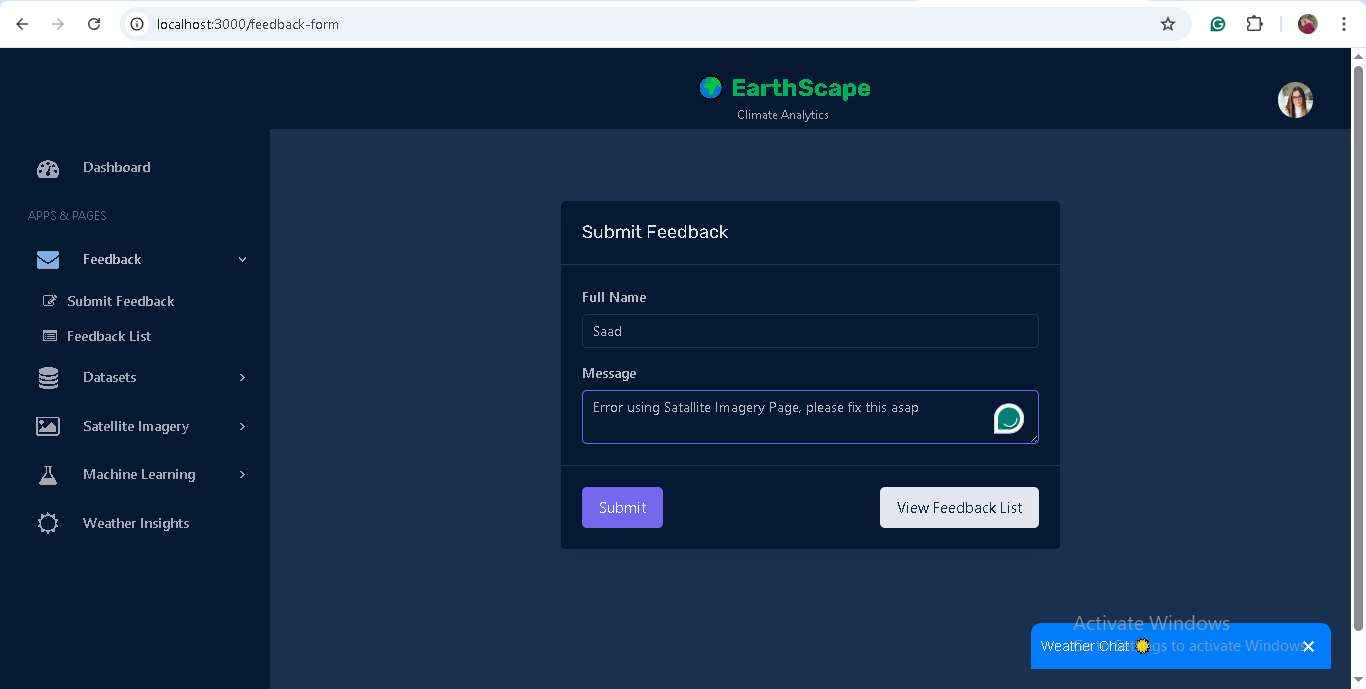
## **1.13 Feedback Management**

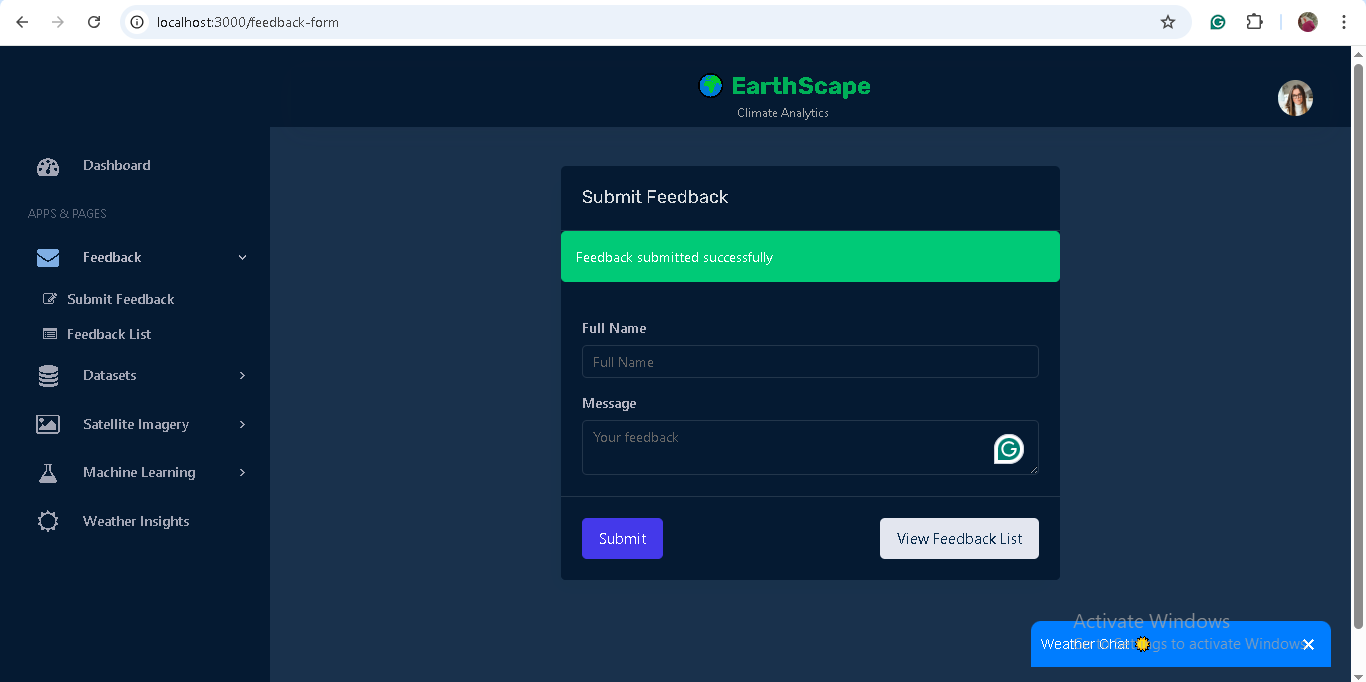
The Feedback module allows users to submit feedback and track its status, while administrators can manage and resolve feedback entries.

### **Features**

**For Analysts (Normal Users):**

* **Submit Feedback**
  + Enter **Full Name** and your **Message**.
  + The system automatically attaches your logged-in email.
  + On success, you see: *“Feedback submitted successfully.”*

**

**

*Code:* from flask import Blueprint, request, jsonify

from db import feedback\_collection

import jwt, os

from datetime import datetime

from bson.objectid import ObjectId

feedback\_bp = Blueprint("feedback\_bp", \_\_name\_\_)

JWT\_SECRET = os.getenv("JWT\_SECRET", "supersecretkey")

JWT\_ALGORITHM = os.getenv("JWT\_ALGORITHM", "HS256")

# Submit feedback (for all users)

@feedback\_bp.route("/feedback", methods=["POST"])

def submit\_feedback():

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    data = request.get\_json()

    user\_email = payload.get("email")

    message = data.get("message", "").strip()

    if not message:

        return jsonify({"error": "Feedback message is required"}), 400

    feedback\_collection.insert\_one({

        "user\_email": user\_email,

        "message": message,

        "status": "pending",

        "created\_at": datetime.utcnow()

    })

    return jsonify({"message": "Feedback submitted successfully"}), 201

# Get all feedback (admin only)

@feedback\_bp.route("/admin-feedback", methods=["GET"])

def admin\_feedback\_list():

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        if payload.get("role") != "admin":

            return jsonify({"error": "Only admins can view feedback"}), 403

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    feedbacks = list(feedback\_collection.find({}, {

        "\_id": 1,

        "user\_email": 1,

        "message": 1,

        "status": 1,

        "created\_at": 1

    }))

    for f in feedbacks:

        f["\_id"] = str(f["\_id"])

    return jsonify(feedbacks), 200

# Get feedback for logged-in user (analyst)

@feedback\_bp.route("/feedback-list", methods=["GET"])

def feedback\_list():

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        user\_email = payload.get("email")

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    feedbacks = list(feedback\_collection.find({"user\_email": user\_email}, {

        "\_id": 1,

        "user\_email": 1,

        "message": 1,

        "status": 1,

        "created\_at": 1

    }))

    for f in feedbacks:

        f["\_id"] = str(f["\_id"])

    return jsonify(feedbacks), 200

# Update feedback status (admin only)

@feedback\_bp.route("/admin-feedback/<feedback\_id>", methods=["PATCH"])

def update\_feedback\_status(feedback\_id):

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        if payload.get("role") != "admin":

            return jsonify({"error": "Only admins can update feedback"}), 403

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    data = request.get\_json()

    status = data.get("status", "").lower()

    if status not in ["pending", "in-progress", "resolved"]:

        return jsonify({"error": "Invalid status"}), 400

    try:

        result = feedback\_collection.update\_one(

            {"\_id": ObjectId(feedback\_id)},

            {"$set": {"status": status}}

        )

    except Exception:

        return jsonify({"error": "Invalid feedback ID"}), 400

    if result.matched\_count == 0:

        return jsonify({"error": "Feedback not found"}), 404

    return jsonify({"message": "Feedback status updated successfully"}), 200

# Delete feedback (admin only)

@feedback\_bp.route("/admin-feedback/<feedback\_id>", methods=["DELETE"])

def delete\_feedback(feedback\_id):

    auth\_header = request.headers.get("Authorization")

    if not auth\_header:

        return jsonify({"error": "Missing token"}), 401

    try:

        token = auth\_header.split(" ")[1]

        payload = jwt.decode(token, JWT\_SECRET, algorithms=[JWT\_ALGORITHM])

        if payload.get("role") != "admin":

            return jsonify({"error": "Only admins can delete feedback"}), 403

    except Exception as e:

        return jsonify({"error": str(e)}), 401

    try:

        result = feedback\_collection.delete\_one({"\_id": ObjectId(feedback\_id)})

    except Exception:

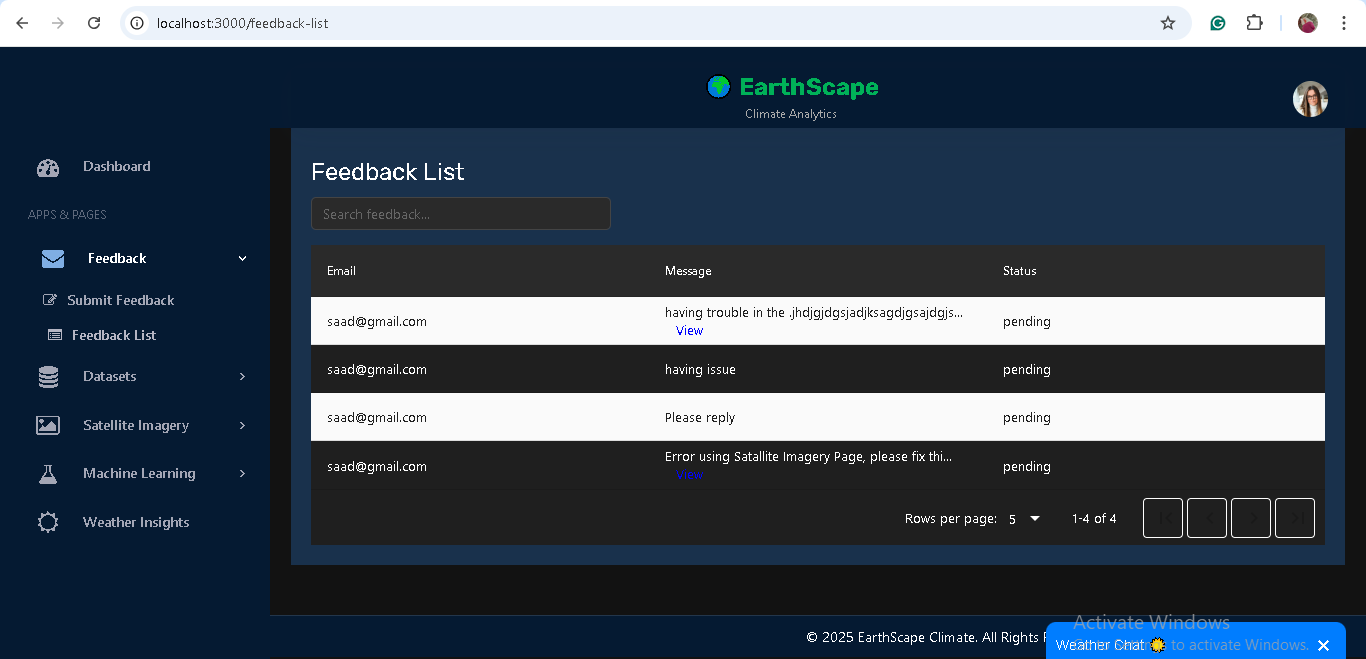
        return jsonify({"error": "Invalid feedback ID"}), 400

    if result.deleted\_count == 0:

        return jsonify({"error": "Feedback not found"}), 404

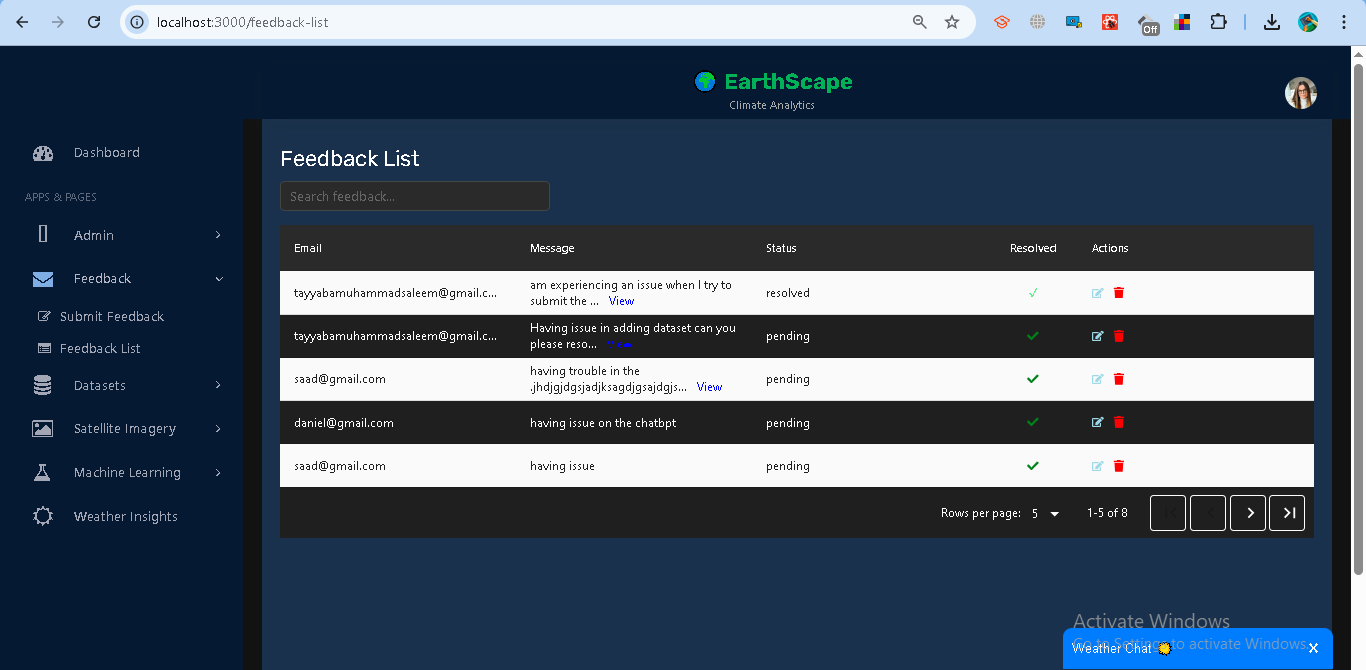
    return jsonify({"message": "Feedback deleted successfully"}), 200

* **View Feedback List**
  + Shows all feedback you have submitted.
  + Displays **message, status, and date submitted**.
  + Status may be *Pending, In-Progress,* or *Resolved*.

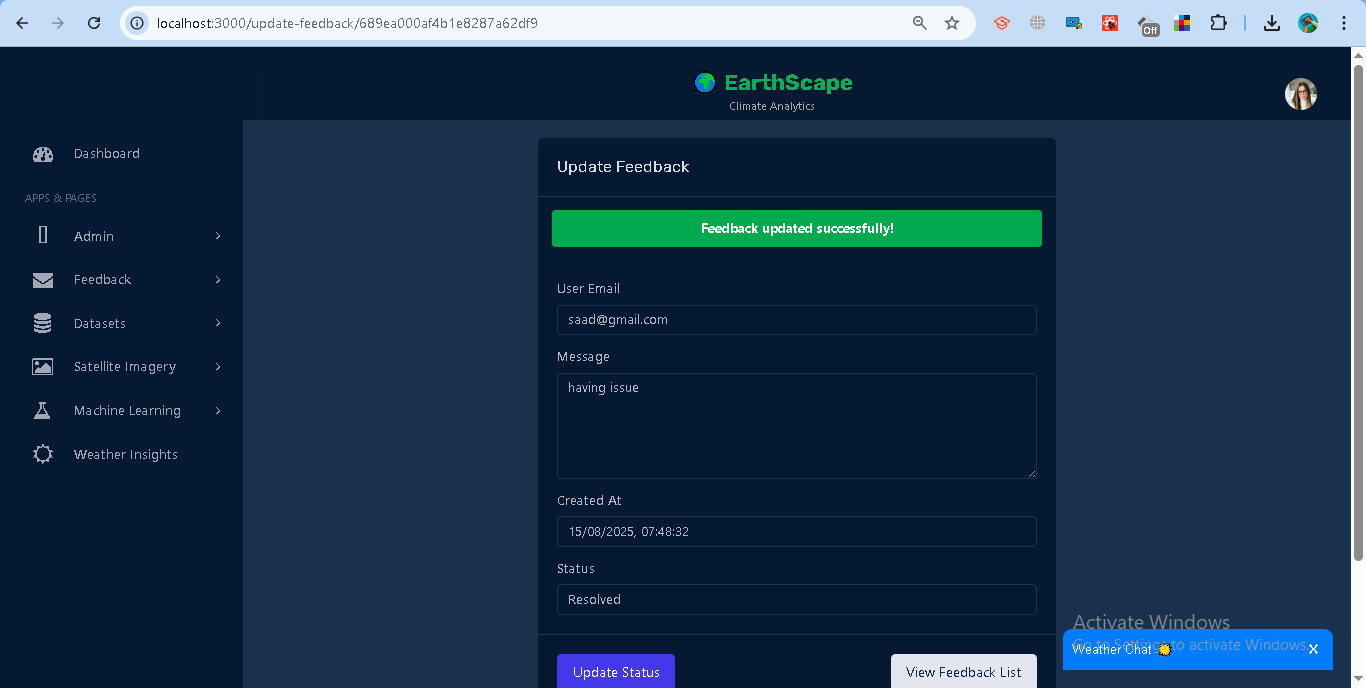


**For Administrators (Admin Role):**

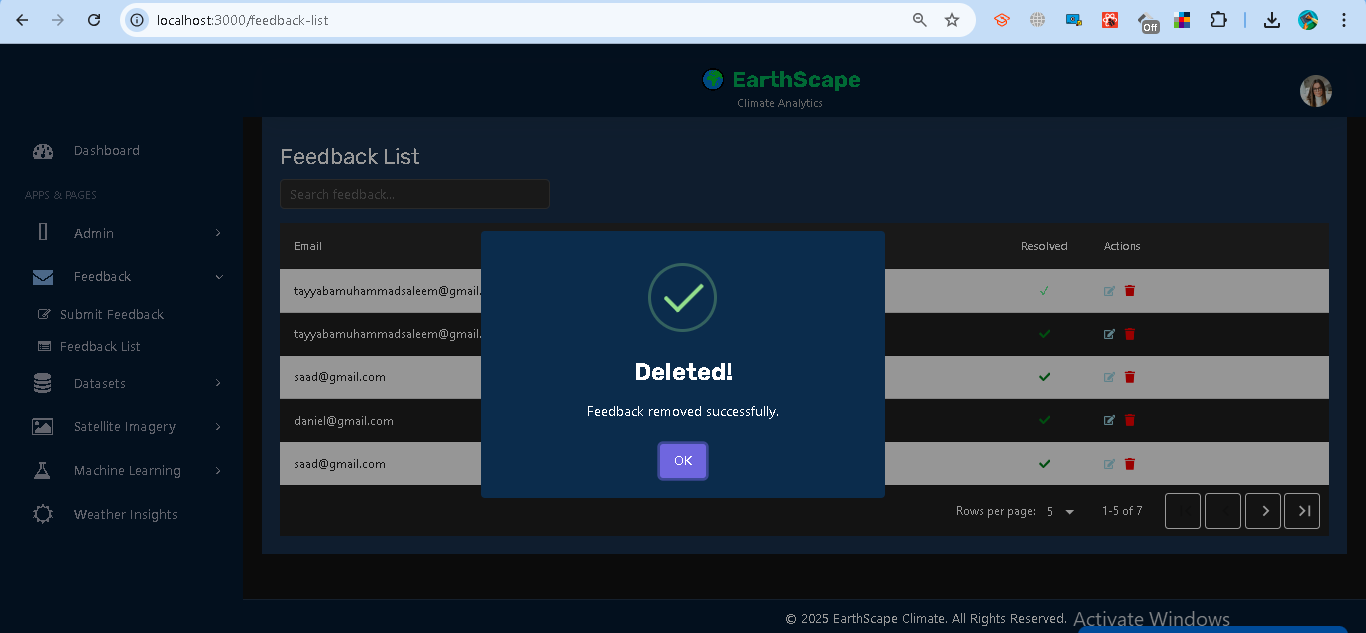
* **View All Feedback**
  + Access to all feedback submitted by any user.
  + Useful for monitoring system-wide feedback.



* **Update Feedback Status**
  + Change status of feedback to *Pending, In-Progress,* or *Resolved*.
  + Helps track progress of reported issues or suggestions.



* **Delete Feedback**
  + Remove feedback entries when necessary.
  + Requires confirmation to prevent accidental deletions.



### **Access Control**

* **Analyst** → Can submit feedback and view their own submissions.
* **Admin** → Full access (view all, update status, delete).

### **Error Handling**

* **Missing Fields** → Prompts users to fill all required fields.
* **Unauthorized** → Requires login to submit or view feedback.
* **Forbidden** → Only admins can manage feedback beyond their own.
* **Invalid ID / Server Errors** → Displayed with appropriate error messages.