Aadhaar Satark: Intelligence Engine

Team: UIDAI_11479

Date: 2026-01-17

Hackathon Track: Societal Trends in Aadhaar Enrolment

1. Problem Statement & Approach

Problem:
The Aadhaar ecosystem handles billions of updates annually. However, gaps exists between 'Expected Biometric Updates' (based on population demographics) and 'Actual Updates'. Identifying these gaps manually across 700+ districts is impossible, leading to potential fraud or service denial.

Approach:
We developed 'Aadhaar Satark', a Real-time Intelligence Engine.
1. Data Ingestion: Integrating Official Data.Gov.in APIs for Enrolment and Biometric data.
2. Smart Merging: A persistent stateful backend that merges monthly datasets without duplication.
3. Anomaly Detection: Implementing an Isolation Forest (Unsupervised ML) model to detect districts where the Update-to-Enrolment ratio deviates significantly from the national norm.
4. Actionable Intelligence: Generating targeted PDF Action Plans for district administrators.

2. Datasets Used

We utilized the official Open Government Data (OGD) Platform APIs provided by UIDAI via Data.Gov.in.

1. Aadhaar Enrolment Data (Resource ID: ecd49b12-3084-4521-8f7e-ca8bf72069ba)
   - Columns Used: State, District, Pincode, Age Group (0-5, 5-18, 18+), Gender.
   - Purpose: To establish the baseline 'Addressable Market' for updates.

2. Aadhaar Biometric Update Data (Resource ID: 65454dab-1517-40a3-ac1d-47d4dfe6891c)
   - Columns Used: State, District, Biometric Updates Count.
   - Purpose: To measure actual performance.

3. Demographic Data (Resource ID: 19eac040-0b94-49fa-b239-4f2fd8677d53)
   - Purpose: Validation of demographic shifts.

3. Methodology

Data Cleaning & Preprocessing:
- Normalization: We implemented a dictionary-based cleaning engine to standardize State/District names (e.g., merging 'Orissa'/'Odisha', correcting 'Westbengal').
- Imputation: Numeric fields are sanitized using reliable zero-filling for missing values.

Analytical Model:
- We calculate the 'Gap Percentage' = (Expected Updates - Actual Updates) / Expected Updates.
- We use Sklearn's Isolation Forest to detect statistical anomalies in this Gap distribution.
- Districts are classified into:
  * CRITICAL (>50% Gap)
  * MODERATE (20-50% Gap)
  * EMERGING (1-20% Gap)
  * COMPLIANT (0% Gap)

4. Analysis & Visualisation

Key Findings from Current Data:
- Total Districts Monitored: 916
- Critical Districts Identified: 36 (Requiring immediate intervention)
- Total Pending Updates Identified: ~42,000

Visualisation:
The solution features a Next.js Dashboard with:
- Interactive Geospatial Map (Color-coded by Risk)
- Real-time Sync Status with Data.Gov.in
- Before/After Trend Analysis

Appendix Code: api_sync.py (Official API Integration)

```
import requests
import pandas as pd
from typing import Dict, List, Optional
import os
from .processing import smart_merge

# Resource IDs from Data.Gov.in
RESOURCES = {
    "biometric": "65454dab-1517-40a3-ac1d-47d4dfe6891c",
    "demographic": "19eac040-0b94-49fa-b239-4f2fd8677d53",
    "enrolment": "ecd49b12-3084-4521-8f7e-ca8bf72069ba"
}

API_KEY = "579b464db66ec23bdd000001cdd3946e44ce4aad7209ff7b23ac571b"
BASE_URL = "https://api.data.gov.in/resource/"

TOTAL_RECORDS_TO_FETCH = 2000  # Fetch enough data to show impact, but don't slow down demo

def fetch_data_gov_resource(resource_id: str, limit: int = 500) -> Optional[pd.DataFrame]:
    """
    Fetches data from a specific Data.Gov.in resource with pagination.
    Fetches up to TOTAL_RECORDS_TO_FETCH records.
    """
    all_records = []
    offset = 0

    print(f"? Connecting to Data.Gov.in Resource: {resource_id}...")

    while len(all_records) < TOTAL_RECORDS_TO_FETCH:
        params = {
            "api-key": API_KEY,
            "format": "json",
            "limit": limit,
            "offset": offset
        }
        url = f"{BASE_URL}{resource_id}"

        try:
            response = requests.get(url, params=params, timeout=10)
            response.raise_for_status()
            data = response.json()

            if 'records' in data and len(data['records']) > 0:
                records = data['records']
                all_records.extend(records)
                print(f"   ? Fetched {len(records)} records (Total: {len(all_records)})")

                if len(records) < limit:
                    break # No more data available

                offset += limit
            else:
                break # No records returned

        except Exception as e:
            print(f"? Error fetching resource {resource_id} at offset {offset}: {e}")
            break
    if all_records:
        df = pd.DataFrame(all_records)
        print(f"? Successfully loaded {len(df)} records from {resource_id}")
        return df

    return None

def sync_all_official_data(master_enrol: pd.DataFrame, master_bio: pd.DataFrame) -> Dict[str, pd.DataFrame]:
    """Syncs all official datasets and merges them into the master dataframes."""
    print("? Starting sync with Data.Gov.in Official Portal...")

    # Enrolment
    new_enrol = fetch_data_gov_resource(RESOURCES["enrolment"])
    if new_enrol is not None:
        master_enrol = smart_merge(master_enrol, new_enrol)
        print(f"? Synced Enrolment: Added/Updated records.")

    # Biometric
    new_bio = fetch_data_gov_resource(RESOURCES["biometric"])
    if new_bio is not None:
        master_bio = smart_merge(master_bio, new_bio)
        print(f"? Synced Biometric: Added/Updated records.")

    return {
        "enrolment": master_enrol,
        "biometric": master_bio
    }
```

Appendix Code: processing.py (ML Logic)

```
import pandas as pd
import numpy as np
from sklearn.ensemble import IsolationForest
import io

# --- GLOBALS: CORRECTION DICTIONARIES ---
STATE_CORRECTIONS = {
    'Orissa': 'Odisha',
    'Uttaranchal': 'Uttarakhand',
    'Pondicherry': 'Puducherry',
    'West Bangal': 'West Bengal',
    'Westbengal': 'West Bengal',
    'Chhatisgarh': 'Chhattisgarh',
    'Tamilnadu': 'Tamil Nadu',
    'Telengana': 'Telangana',
    'Andaman & Nicobar Islands': 'Andaman And Nicobar Islands',
    'Dadra & Nagar Haveli': 'Dadra And Nagar Haveli And Daman And Diu',
    'The Dadra And Nagar Haveli And Daman And Diu': 'Dadra And Nagar Haveli And Daman And Diu',
    'Dadra And Nagar Haveli': 'Dadra And Nagar Haveli And Daman And Diu',
    'Daman & Diu': 'Dadra And Nagar Haveli And Daman And Diu',
    'Daman And Diu': 'Dadra And Nagar Haveli And Daman And Diu',
    'Jammu & Kashmir': 'Jammu And Kashmir',
}

DISTRICT_CORRECTIONS = {
    'Visakhapatanam': 'Visakhapatnam',
    'Spsr Nellore': 'Nellore',
    'Sri Potti Sriramulu Nellore': 'Nellore',
    'Y. S. R.': 'Ysr',
    'Y.S.R.': 'Ysr',
    'Y. S. R': 'Ysr',
    'K.V. Rangareddy': 'Ranga Reddy',
    'K.V.Rangareddy': 'Ranga Reddy',
    'Rangareddi': 'Ranga Reddy',
    'Mahabub Nagar': 'Mahabubnagar',
    'Mahbubnagar': 'Mahabubnagar',
    'East Midnapore': 'Purba Medinipur',
    'East Midnapur': 'Purba Medinipur',
    'West Midnapore': 'Paschim Medinipur',
    'West Medinipur': 'Paschim Medinipur',
    'Coochbehar': 'Cooch Behar',
    'Koch Bihar': 'Cooch Behar',
    'Hooghly': 'Hugli',
    'Hooghiy': 'Hugli',
    'Howrah': 'Haora',
    'Hawrah': 'Haora',
    'Burdwan': 'Purba Bardhaman',
    'Bardhaman': 'Purba Bardhaman',
    'Sibsagar': 'Sivasagar',
    'Panch Mahals': 'Panchmahal',
    'Panchmahals': 'Panchmahal',
    'Dohad': 'Dahod',
    'Kachchh': 'Kutch',
    'Ahmadabad': 'Ahmedabad',
    'Baramula': 'Baramulla',
    'Bandipore': 'Bandipora',
    'Badgam': 'Budgam',
    'Shupiyan': 'Shopian',
    'S.A.S Nagar': 'Sas Nagar',
    'S.A.S Nagar(Mohali)': 'Sas Nagar',
    'Sas Nagar (Mohali)': 'Sas Nagar',
    'Ferozepur': 'Firozpur',
    'Bathinda': 'Bhatinda',
    'Muktsar': 'Sri Muktsar Sahib',
    'Mewat': 'Nuh',
    'Gurgaon': 'Gurugram',
    'Yamuna Nagar': 'Yamunanagar',
    'Purbi Champaran': 'East Champaran',
    'Purba Champaran': 'East Champaran',
    'Pashchim Champaran': 'West Champaran',
    'Kaimur (Bhabua)': 'Kaimur',
    'Bhabua': 'Kaimur',
    'Jehanabad': 'Jahanabad',
    'Aurangabad(Bh)': 'Aurangabad',
    'Sant Ravidas Nagar': 'Bhadohi',
    'Sant Ravidas Nagar Bhadohi': 'Bhadohi',
    'Kushi Nagar': 'Kushinagar',
    'Mahrajganj': 'Maharajganj',
    'Siddharth Nagar': 'Siddharthnagar',
    'Shrawasti': 'Shravasti',
    'Bara Banki': 'Barabanki',
    'Lakhimpur Kheri': 'Kheri',
    'Jyotiba Phule Nagar': 'Amroha',
    'Nainital': 'Naini Tal',
    'Hardwar': 'Haridwar',
    'Pauri Garhwal': 'Garhwal',
    'Tehri Garhwal': 'Tehri',
    'Udham Singh Nagar': 'Udham Singh Nagar',
    'South 24 Parganas': 'South Twenty Four Parganas',
    'North 24 Parganas': 'North Twenty Four Parganas',
    'South 24 Pargana': 'South Twenty Four Parganas',
    'Dakshin Dinajpur': 'South Dinajpur',
    'Uttar Dinajpur': 'North Dinajpur',
    'Dinajpur Dakshin': 'South Dinajpur',
    'Dinajpur Uttar': 'North Dinajpur',
    'Darjiling': 'Darjeeling',
    'Maldah': 'Malda',
    'Puruliya': 'Purulia',
    'Siwan': 'Sewan',
    'Saran': 'Chapra',
}

# --- HELPER FUNCTIONS ---

def clean_dataframe(df: pd.DataFrame) -> pd.DataFrame:
    """
    Applies standard cleaning and normalization to a dataframe.
    """
    if df is None or df.empty:
        return df

    # Standardize column names
    df.columns = [c.lower().strip() for c in df.columns]

    # Filter out numeric or invalid districts
    def is_valid_name(name):
        return isinstance(name, str) and not name.isdigit() and len(name) > 1

    if 'district' in df.columns:
        df = df[df['district'].apply(is_valid_name)].copy()
    # Normalize Text (State/District)
    for col in ['state', 'district']:
        if col in df.columns:
            df[col] = df[col].astype(str).str.strip().replace(r'\s+', ' ', regex=True).str.title()

    # Apply State Corrections
    if 'state' in df.columns:
        df['state'] = df['state'].replace(STATE_CORRECTIONS)

    # Apply District Corrections
    def clean_district_name(name):
        name = name.split('(')[0].split('*')[0].strip()
        return DISTRICT_CORRECTIONS.get(name, name)

    if 'district' in df.columns:
        df['district'] = df['district'].apply(clean_district_name)

    return df


def smart_merge(existing_df: pd.DataFrame, new_df: pd.DataFrame) -> pd.DataFrame:
    """
    Merges new data with existing data, handling deduplication.
    Deduplication Policy:
    - Keys: State, District, Pincode, Date (if available)
    - Conflict: Keep LAST (Assuming new upload is the latest source of truth for that record)
    """
    # 1. Clean the new data to match the standard format of existing data
    new_df = clean_dataframe(new_df)

    if existing_df is None or existing_df.empty:
        return new_df

    # 2. Concatenate
    combined = pd.concat([existing_df, new_df], ignore_index=True)

    # 3. Define Deduplication Subset
    # We want to identify if a specific record (location + time) is being re-uploaded
    subset = ['state', 'district', 'pincode']
    if 'date' in combined.columns:
        subset.append('date')

    # Remove columns that might not exist in subset for robustness (though we cleaned)
    subset = [c for c in subset if c in combined.columns]

    # 4. Drop Duplicates (Keep Last = New Upload Overwrites Old)
    combined.drop_duplicates(subset=subset, keep='last', inplace=True)

    return combined


def process_data(enrolment_data, biometric_data, demographic_data=None, model=None):
    try:
        # 1. Load Data (Handle Bytes or DataFrame)
        if isinstance(enrolment_data, pd.DataFrame):
            df_enrolment = enrolment_data.copy() # Copy to avoid mutating input if cached
        else:
            df_enrolment = pd.read_csv(io.BytesIO(enrolment_data))

        if isinstance(biometric_data, pd.DataFrame):
            df_biometric = biometric_data.copy()
        else:
            df_biometric = pd.read_csv(io.BytesIO(biometric_data))

        if demographic_data is not None:
             if isinstance(demographic_data, pd.DataFrame):
                df_demographic = demographic_data.copy()
             else:
                df_demographic = pd.read_csv(io.BytesIO(demographic_data))
        else:
            df_demographic = pd.DataFrame(columns=['state', 'district', 'demo_age_5_17'])

        # 2. Clean Data (Using centralized logic)
        # Note: If called from main.py's smart_merge, data is already cleaned.
        # But running it again is safe (idempotent) and ensures consistency if called directly.
        df_enrolment = clean_dataframe(df_enrolment)
        df_biometric = clean_dataframe(df_biometric)
        df_demographic = clean_dataframe(df_demographic)

        # Convert numeric columns to proper types (handles API data that comes as strings)
        for col in ['age_5_17', 'bio_age_5_17', 'demo_age_5_17']:
            if col in df_enrolment.columns:
                df_enrolment[col] = pd.to_numeric(df_enrolment[col], errors='coerce').fillna(0)
            if col in df_biometric.columns:
                df_biometric[col] = pd.to_numeric(df_biometric[col], errors='coerce').fillna(0)
            if col in df_demographic.columns:
                df_demographic[col] = pd.to_numeric(df_demographic[col], errors='coerce').fillna(0)

        # 3. Aggregate by District
        # We group by State+District to get the Total Counts for the Dashboard
        grp_enrolment = df_enrolment.groupby(['state', 'district'])['age_5_17'].sum().reset_index()
        grp_biometric = df_biometric.groupby(['state', 'district'])['bio_age_5_17'].sum().reset_index()

        if not df_demographic.empty:
                                                  grp_demographic    =    df_demographic.groupby(['state',
'district'])['demo_age_5_17'].sum().reset_index()
             grp_demographic.rename(columns={'demo_age_5_17': 'demo_updates'}, inplace=True)
        else:
             grp_demographic = pd.DataFrame(columns=['state', 'district', 'demo_updates'])

        # 4. Merge Aggregates
        merged = pd.merge(grp_enrolment, grp_biometric, on=['state', 'district'], how='outer').fillna(0)

        if not grp_demographic.empty:
            merged = pd.merge(merged, grp_demographic, on=['state', 'district'], how='left').fillna(0)
        else:
            merged['demo_updates'] = 0

        # 5. Calculate Metrics
        merged['expected_updates'] = merged['age_5_17']
        merged['actual_updates'] = merged['bio_age_5_17']
        merged['pending_updates'] = merged['expected_updates'] - merged['actual_updates']
          merged['gap_percentage'] = (merged['pending_updates'] / merged['expected_updates']).replace([np.inf,
-np.inf], 0).fillna(0) * 100

        merged['pending_updates'] = merged['pending_updates'].clip(lower=0)
        merged['gap_percentage'] = merged['gap_percentage'].clip(lower=0, upper=100)

        # 6. Anomaly Detection Rules
        features = ['pending_updates', 'gap_percentage', 'demo_updates']

        if len(merged) > 1:
            if model is None:
```

```
            # TRAIN MODE
            model = IsolationForest(contamination=0.1, random_state=42)
            merged['anomaly_score'] = model.fit_predict(merged[features])
        else:
            # PREDICT MODE
            try:
                merged['anomaly_score'] = model.predict(merged[features])
            except ValueError:
                # Fallback if model was trained with fewer features (backward compatibility)
                print("?? Model feature mismatch. Falling back to 2 features.")
                merged['anomaly_score'] = model.predict(merged[['pending_updates', 'gap_percentage']])

        merged['is_anomaly'] = merged['anomaly_score'] == -1
    else:
        merged['is_anomaly'] = False

    # 7. Formatting Output
    districts_data = []
    for _, row in merged.iterrows():
        status = "SAFE"
        reason = ""

        if row['gap_percentage'] > 50:
            status = "CRITICAL"
             reason = "High Deficit Alert: Over 50% gap indicates immediate intervention needed. Possible
migration hub or lack of centers."
        elif row['gap_percentage'] > 20:
            status = "MODERATE"
            reason = "Warning: Gap is widening. Schedule camps to prevent backlog accumulation."
        else:
            reason = "Normal operations. Updates usage consistent with enrolment."

        if row['is_anomaly']:
             if status == "SAFE":
                reason = "Unusual Pattern Detected: Metric outlier despite safe status."
             else:
                reason += " [AI Anomaly]: Statistical outlier detected relative to state patterns."

             if row['demo_updates'] > 0 and abs(row['demo_updates'] - row['actual_updates']) > 1000:
                 diff = int(row['demo_updates'] - row['actual_updates'])
                 reason += f" High variance seen in demographic data ({diff} difference)."

        districts_data.append({
            "state": row['state'],
            "district": row['district'],
            "expected_updates": int(row['expected_updates']),
            "actual_updates": int(row['actual_updates']),
            "pending_updates": int(row['pending_updates']),
            "gap_percentage": round(row['gap_percentage'], 1),
            "status": status,
            "is_anomaly": bool(row['is_anomaly']),
            "ai_reasoning": reason
        })

    total_pending = int(merged['pending_updates'].sum())
    critical_count = int(merged[merged['gap_percentage'] > 50].shape[0])

    return {
        "summary": {
            "total_pending_updates": total_pending,
            "critical_districts_count": critical_count,
            "processed_districts": len(merged)
            },
            "districts": districts_data,
            "model": model
        }

    except Exception as e:
        print(f"Error processing data: {e}")
        return {"error": str(e)}

if __name__ == "__main__":
    pass
```

Appendix Code: main.py (FastAPI Backend)

```
from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse, Response
from pydantic import BaseModel
import uvicorn
import io
import pandas as pd
import joblib
import json
import os
import shutil

# Import from refactored processing module
from services.processing import process_data, smart_merge
from services.report_generator import generate_report
from services.api_sync import sync_all_official_data

app = FastAPI(title="Aadhaar Satark API")

# Setup CORS
origins = [
    "http://localhost:3000",
    "http://127.0.0.1:3000",
    "http://localhost:3001",
    "http://127.0.0.1:3001",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

class DistrictReportRequest(BaseModel):
    state: str
    district: str
    expected_updates: int
    actual_updates: int
    pending_updates: int
    gap_percentage: float
    status: str

@app.get("/health")
def health_check():
    """Simple health check endpoint."""
    return {"status": "ok"}

# Globals for Persistence
TRAINED_MODEL = None
GLOBAL_ENROL_DF = None
GLOBAL_BIO_DF = None
GLOBAL_DEMO_DF = None

DATA_DIR = "data"
ENROL_PATH = os.path.join(DATA_DIR, "master_enrolment.pkl")
BIO_PATH = os.path.join(DATA_DIR, "master_biometric.pkl")
```

```
DEMO_PATH = os.path.join(DATA_DIR, "master_demographic.pkl")
MODEL_PATH = "models/isolation_forest.joblib"
INITIAL_DATA_PATH = "data/initial_data.json"

@app.on_event("startup")
async def load_artifacts():
    global TRAINED_MODEL, GLOBAL_ENROL_DF, GLOBAL_BIO_DF, GLOBAL_DEMO_DF
    try:
        os.makedirs(DATA_DIR, exist_ok=True)

        # 1. Load Model
        if os.path.exists(MODEL_PATH):
            print(f"? Loading Trained Model from {MODEL_PATH}...")
            TRAINED_MODEL = joblib.load(MODEL_PATH)

        # 2. Load Master Datasets (if they exist)
        if os.path.exists(ENROL_PATH):
            print(f"? Loading Enrolment Master from {ENROL_PATH}...")
            try:
                GLOBAL_ENROL_DF = pd.read_pickle(ENROL_PATH)
            except Exception as e:
                print(f"?? Error loading Enrolment Master: {e}")

        if os.path.exists(BIO_PATH):
            print(f"? Loading Biometric Master from {BIO_PATH}...")
            try:
                GLOBAL_BIO_DF = pd.read_pickle(BIO_PATH)
            except Exception as e:
                print(f"?? Error loading Biometric Master: {e}")

        if os.path.exists(DEMO_PATH):
            print(f"? Loading Demographic Master from {DEMO_PATH}...")
            try:
                GLOBAL_DEMO_DF = pd.read_pickle(DEMO_PATH)
            except Exception as e:
                print(f"?? Error loading Demographic Master: {e}")

        print("? System Initialization Complete. Persistence Layer Active.")

    except Exception as e:
        print(f"?? Warning: Failed to load artifacts: {e}")

def save_state():
    """Helper to save current global DFs to disk"""
    try:
        if GLOBAL_ENROL_DF is not None:
            GLOBAL_ENROL_DF.to_pickle(ENROL_PATH)
        if GLOBAL_BIO_DF is not None:
            GLOBAL_BIO_DF.to_pickle(BIO_PATH)
        if GLOBAL_DEMO_DF is not None:
            GLOBAL_DEMO_DF.to_pickle(DEMO_PATH)
        print("? State saved successfully.")
    except Exception as e:
        print(f"? Error Saving State: {e}")

@app.get("/initial-data")
async def get_initial_data():
    """
    Returns the processed analysis.
    If Global DFs are loaded, calculate fresh from them.
    Else fall back to initial_data.json.
    """
    # Priority: Real-time Global Data > Static JSON
    if GLOBAL_ENROL_DF is not None and GLOBAL_BIO_DF is not None:
        try:
            print("? Generating fresh insights from Persistent Store...")
            result = process_data(
                GLOBAL_ENROL_DF,
                GLOBAL_BIO_DF,
                demographic_data=GLOBAL_DEMO_DF,
                model=TRAINED_MODEL
            )
            if "model" in result: result.pop("model")

            # Add metadata
            result['dataset_info'] = {
                "enrolment_records": len(GLOBAL_ENROL_DF),
                "biometric_records": len(GLOBAL_BIO_DF),
                "source": "persistent_store"
            }
            return JSONResponse(content=result)
        except Exception as e:
            print(f"?? generation error: {e}. Falling back to static file.")

    # Fallback
    if os.path.exists(INITIAL_DATA_PATH):
        with open(INITIAL_DATA_PATH, "r") as f:
            return JSONResponse(content=json.load(f))

    return {"error": "No data available. Please upload files or run training."}

@app.post("/sync-official")
async def sync_official():
    global GLOBAL_ENROL_DF, GLOBAL_BIO_DF

    # Initialize if None (Fallbacks to handle empty start)
    if GLOBAL_ENROL_DF is None: GLOBAL_ENROL_DF = pd.DataFrame()
    if GLOBAL_BIO_DF is None: GLOBAL_BIO_DF = pd.DataFrame()

    try:
        results = sync_all_official_data(GLOBAL_ENROL_DF, GLOBAL_BIO_DF)
        GLOBAL_ENROL_DF = results["enrolment"]
        GLOBAL_BIO_DF = results["biometric"]

        save_state()

        return {
            "status": "success",
            "message": "Official Data Synced successfully",
            "enrolment_size": len(GLOBAL_ENROL_DF),
            "biometric_size": len(GLOBAL_BIO_DF)
        }
    except Exception as e:
        print(f"? Sync Error: {e}")
        raise HTTPException(status_code=500, detail=f"Sync failed: {str(e)}")

@app.post("/upload")
async def upload_files(
    enrolment_file: UploadFile = File(None),
    biometric_file: UploadFile = File(None),
    demographic_file: UploadFile = File(None)
):
    global GLOBAL_ENROL_DF, GLOBAL_BIO_DF, GLOBAL_DEMO_DF
    import time
```

```
start_time = time.time()
try:
    # 1. Read Uploaded Files (Handle Partial Uploads)
    # Note: 'File' default is not None for File(...), so we check if provided

    updated = False

    if enrolment_file:
        print("? Processing Enrolment Update...")
        enrol_bytes = await enrolment_file.read()
        if len(enrol_bytes) > 0:
            new_enrol_df = pd.read_csv(io.BytesIO(enrol_bytes))
            GLOBAL_ENROL_DF = smart_merge(GLOBAL_ENROL_DF, new_enrol_df)
            updated = True

    if biometric_file:
         print("? Processing Biometric Update...")
         bio_bytes = await biometric_file.read()
         if len(bio_bytes) > 0:
            new_bio_df = pd.read_csv(io.BytesIO(bio_bytes))
            GLOBAL_BIO_DF = smart_merge(GLOBAL_BIO_DF, new_bio_df)
            updated = True

    if demographic_file:
         print("? Processing Demographic Update...")
         demo_bytes = await demographic_file.read()
         if len(demo_bytes) > 0:
            new_demo_df = pd.read_csv(io.BytesIO(demo_bytes))
            GLOBAL_DEMO_DF = smart_merge(GLOBAL_DEMO_DF, new_demo_df)
            updated = True

    if not updated:
        return JSONResponse({"message": "No valid files received or empty files."}, status_code=400)

    # 2. Save State (Persistence)
    save_state()

    # 3. Process (Run Analysis on Global Data)
    result = process_data(
        GLOBAL_ENROL_DF,
        GLOBAL_BIO_DF,
        demographic_data=GLOBAL_DEMO_DF,
        model=TRAINED_MODEL
    )

    if "model" in result:
        result.pop("model")

    if "error" in result:
         raise HTTPException(status_code=400, detail=result["error"])

    # Latency
    end_time = time.time()
    result['processing_time_ms'] = round((end_time - start_time) * 1000, 2)

    result['dataset_info'] = {
        "enrolment_records": len(GLOBAL_ENROL_DF) if GLOBAL_ENROL_DF is not None else 0,
        "biometric_records": len(GLOBAL_BIO_DF) if GLOBAL_BIO_DF is not None else 0,
         "source": "live_update"
    }

    return JSONResponse(content=result)
    except Exception as e:
        import traceback
        traceback.print_exc()
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/generate-report")
async def generate_pdf(report_request: DistrictReportRequest):
    try:
        data = report_request.dict()
        pdf_bytes = generate_report(data)

        headers = {
            'Content-Disposition': f'attachment; filename="Report_{data["district"]}.pdf"'
        }
        return Response(content=pdf_bytes, media_type="application/pdf", headers=headers)
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

if __name__ == "__main__":
    # Use port 8001 to avoid conflicts if 8000 is taken, or match user env
    uvicorn.run("main:app", host="0.0.0.0", port=8001, reload=True)
```