Designing an AI-based platform that integrates:

1. multiple datasets,
2. applies machine learning,
3. and supports dynamic querying via a natural language interface.

**1. Initial Thoughts on Platform Architecture**

**Goal**: Design a scalable backend architecture that meets NCIF’s objectives.

**Key Considerations:**

* **Data Integration**: The platform will integrate 21 datasets, including structured and unstructured data.
* **Practices**:
  + **Scalable Infrastructure**: Utilize AWS EMR🡪EC2 for computing, S3 for data storage, and RDS for database management.
  + **Security and Authentication**: Implement encryption using Key Management Services AWS, IAM role (accessibility to the services) for access control and use Amazon Cognito for user authentication (recognize accessed person).
  + **Large Language Models (LLMs)**: Deploy an LLM on AWS SageMaker (AWS secret manager for API caching) for NLP tasks.

**High-Level Architecture:**

* **Data Layer**:
  + Store structured data in AWS RDS or Amazon Aurora (for SQL-based data like SOD, NCUA).
  + Store unstructured data in AWS S3 (for documents, images, etc.).
  + Use AWS ~~Glue~~ EMR to clean, transform, and load data into the database.
* **Backend API Layer**:
  + Develop RESTful APIs using AWS Lambda for serverless computers and API Gateway for managing endpoints.
  + Use AWS S3 ~~DynamoDB or RDS~~ for storing temporary results or cache for quick retrieval.
* **Security & Authentication**:
  + Use **Amazon Cognito** for handling user authentication.
  + Use AWS IAM roles to manage permissions for different user groups.
* **NLP task handling Layer**:
  + Use **Amazon SageMaker** to train and deploy machine learning models or integrate third-party LLM APIs like OpenAI’s GPT models for NLP tasks.
* **Data Retrieval & Querying**:
  + Use **Amazon Athena** to run queries directly on S3-stored unstructured data.
  + Use **AWS Lambda** for dynamically querying the database based on user inputs.

**Architecture Diagram Summary**:

* The diagram includes a data pipeline starting from data ingestion (S3 and RDS) through AWS Glue for transformation, stored in RDS/Aurora, and accessible via API Gateway. The user interfaces (web or mobile) would interact with the API layer, which in turn uses Lambda and DynamoDB for real-time processing and querying.

**Task 2:**

**Data Integration: Load Data into SQLite**

Load datasets from AWS S3 into an SQLite database using EMR Spark.

This process ensures efficient handling of data stored in AWS S3, transforming it into a format suitable for relational databases (SQLite). Using Apache Spark ensures scalability and the ability to handle large volumes of data.

**Steps**

**1. Set Up AWS EMR Cluster with Spark**

1.1 **Launch AWS EMR Cluster**

* Access the AWS Management Console and launch an EMR cluster.
* Enable **Apache Spark** in the configuration to handle large-scale data processing.
* Attach an **IAM role** with access permissions to the S3 bucket storing the datasets.

1.2 **Cluster Configuration**

* Install necessary libraries on the cluster, including:
  + **JDBC drivers** for SQLite.
  + Additional Python libraries (pandas, sqlite3, etc.)
* Configure the cluster security group to allow secure communication with the on-premise SQLite system.

**Step 2: Read Data from AWS S3**

* Use **PySpark** to read the datasets stored in S3.
* Ensure data integrity by verifying the schema and contents of each dataset.

from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder \

.appName("DataIntegration") \

.getOrCreate()

# Read datasets from S3 bucket

epa\_data = spark.read.csv("s3://bucket-name/EPA\_Air\_Quality.csv", header=True, inferSchema=True)

sod\_data = spark.read.csv("s3://bucket-name/SOD.csv", header=True, inferSchema=True)

ncua\_data = spark.read.csv("s3://bucket-name/NCUA.csv", header=True, inferSchema=True)

**Step 3: Data Transformation with Spark**

* **Normalize Data**: Ensure column names and types are consistent across datasets.
* **Validate Master Keys**:
  + Verify census\_tract exists in all datasets for geospatial connections.
  + Ensure CERT exists in **SOD** and **NCUA** datasets for operational connections.

# Standardize column names for consistency

epa\_data = epa\_data.withColumnRenamed("CensusTract", "census\_tract")

sod\_data = sod\_data.withColumnRenamed("FDIC\_Cert", "cert").withColumnRenamed("CensusTract", "census\_tract")

ncua\_data = ncua\_data.withColumnRenamed("CERT", "cert").withColumnRenamed("CensusTract", "census\_tract")

# Join datasets based on Master Keys

geospatial\_join = sod\_data.join(ncua\_data, on="cert", how="inner") # Operational connection

integrated\_data = geospatial\_join.join(epa\_data, on="census\_tract", how="inner") # Geospatial connection

**Step 4: Save Transformed Data to SQLite**

* **Install JDBC Driver**: Ensure the cluster has an SQLite JDBC driver installed to allow PySpark to write directly to SQLite.
* **Write Data**: Store each dataset into its respective table in SQLite, ensuring that key relationships are preserved.

# Write transformed data to SQLite

integrated\_data.write \

.format("jdbc") \

.option("url", "jdbc:sqlite:/path/to/on-premise-database.sqlite") \

.option("dbtable", "IntegratedData") \

.option("driver", "org.sqlite.JDBC") \

.mode("overwrite") \

.save()

**Step 5: Verify Data Integration**

1. **Query SQLite Tables**: Confirm the successful load of data into the SQLite database using Python's sqlite3 library.

import sqlite3

# Connect to the SQLite database

conn = sqlite3.connect("/path/to/on-premise-database.sqlite")

cursor = conn.cursor()

# Verify tables

cursor.execute("SELECT COUNT(\*) FROM IntegratedData")

print(f"Total Rows in IntegratedData Table: {cursor.fetchone()[0]}")

conn.close()

1. **Ensure Key Integrity**:

* Run checks to ensure the integrity of census\_tract and cert keys across tables.

Task 4. API Creation

To create the APIs described in your task, I used several steps to build a solution that aggregates data, supports dynamic querying, integrates machine learning for predictions, and processes natural language input. Here’s an overview of how each API was created and how they work:

**1. API for Data Aggregation**

The goal here is to aggregate data from multiple sources (e.g., SOD, NCUA, and EPA datasets) and compute branch density by Census Tract, while categorizing air quality levels based on **PM2.5** data.

**Steps Taken:**

* **Flask Setup**: Flask was used to build the API because it’s lightweight and easy to integrate with SQL databases.
* **Database Setup**: Assumed the use of **SQLite** (but can be replaced with PostgreSQL, MySQL, etc.). Data from branches (SOD and NCUA) and air quality (EPA) are assumed to be stored in relevant tables.
* **SQL Query**: A SQL query was written to join the branches\_data and air\_quality\_data tables on the common census\_tract key, then calculate branch density and categorize air quality based on **PM2.5** levels.

**Code Example:**

from flask import Flask, jsonify

import sqlite3

app = Flask(\_\_name\_\_)

def get\_db\_connection():

conn = sqlite3.connect('data.db')

return conn

@app.route('/aggregate\_data', methods=['GET'])

def aggregate\_data():

conn = get\_db\_connection()

cursor = conn.cursor()

query = """

SELECT census\_tract, COUNT(branch\_id) AS branch\_count,

(COUNT(branch\_id) / area) AS branch\_density,

CASE

WHEN pm25 <= 12 THEN 'Low'

WHEN pm25 <= 35 THEN 'Moderate'

ELSE 'High'

END AS air\_quality

FROM branches\_data

JOIN air\_quality\_data ON branches\_data.census\_tract = air\_quality\_data.census\_tract

GROUP BY census\_tract;

"""

cursor.execute(query)

data = cursor.fetchall()

conn.close()

return jsonify(data)

**How it works:**

* **Census Tract Aggregation**: For each Census Tract, we count the branches (both bank and credit union), compute their density, and categorize the air quality based on **PM2.5** levels (Low, Moderate, High).
* **Response**: The results are returned in JSON format for easy consumption.

**2. API for Dynamic Querying**

The goal here is to allow users to input conditions (like PM2.5 above 15 and more than 5 branches) and dynamically fetch results from the database based on the input.

**Steps Taken:**

* **User Input Parsing**: The user sends input via query parameters (e.g., ?pm25\_threshold=15&branch\_count\_threshold=5).
* **SQL Query Construction**: The input values are used to dynamically construct a SQL query to filter results accordingly.
* **SQL Execution**: The SQL query is executed against the database, and the results are returned as JSON.

**Code Example:**

@app.route('/query', methods=['GET'])

def query\_data():

pm25\_threshold = float(request.args.get('pm25\_threshold', 15)) # Default value of 15

branch\_threshold = int(request.args.get('branch\_count\_threshold', 5)) # Default value of 5

query = f"""

SELECT census\_tract, branch\_count, pm25

FROM census\_data

WHERE pm25 > {pm25\_threshold} AND branch\_count > {branch\_threshold};

"""

conn = get\_db\_connection()

cursor = conn.cursor()

cursor.execute(query)

data = cursor.fetchall()

conn.close()

return jsonify(data)

**How it works:**

* **Dynamic Filtering**: The API parses the query parameters, and based on those parameters, constructs a dynamic SQL query.
* **Flexible Input**: Users can adjust the pm25\_threshold and branch\_count\_threshold to filter the data based on their needs.

**3. Machine Learning API for PM2.5 Prediction**

The goal here is to use historical data to train a **machine learning model** that predicts **PM2.5 levels** based on branch density and historical pollution data.

**Steps Taken:**

* **Data Preprocessing**: Data from historical pollution and branch density is collected and prepared for training the machine learning model.
* **Model Training**: A simple **Linear Regression** model is used for prediction, based on branch density and historical pollution data.
* **Prediction API**: Users can send data (branch density and historical PM2.5) and receive predictions on future PM2.5 levels.

**Code Example:**

from sklearn.linear\_model import LinearRegression

from flask import Flask, request, jsonify

import pandas as pd

app = Flask(\_\_name\_\_)

# Example historical data (simplified)

data = pd.DataFrame({

'branch\_density': [5, 10, 15, 20],

'historical\_pm25': [12, 15, 17, 20],

'pm25': [10, 16, 18, 21]

})

# Split data

X = data[['branch\_density', 'historical\_pm25']]

y = data['pm25']

# Train the model

model = LinearRegression()

model.fit(X, y)

@app.route('/predict\_pm25', methods=['POST'])

def predict\_pm25():

data = request.get\_json() # Expecting input JSON like {"branch\_density": 12, "historical\_pm25": 14}

branch\_density = data['branch\_density']

historical\_pm25 = data['historical\_pm25']

# Make prediction

prediction = model.predict([[branch\_density, historical\_pm25]])

return jsonify({'predicted\_pm25': prediction[0]})

**How it works:**

* **Model Training**: The API trains a **Linear Regression** model using historical data to predict **PM2.5 levels** based on branch density and historical pollution data.
* **Prediction**: The model predicts **PM2.5** levels based on user-provided input (branch density and historical PM2.5).

**4. NLP Interface for Query Interpretation**

The goal here is to allow users to input natural language queries (e.g., “Show me all tracts with above-average air pollution and a bank branch”) and have the API convert those queries into SQL queries.

**Steps Taken:**

* **Natural Language Processing**: I used **spaCy**, a Python library for **Named Entity Recognition (NER)**, to parse the user's natural language query and extract relevant entities (like PM2.5 thresholds and branch counts).
* **SQL Query Construction**: Based on the extracted entities, the API constructs a dynamic SQL query and returns the results.

**Code Example:**

import spacy

from flask import Flask, request, jsonify

nlp = spacy.load('en\_core\_web\_sm')

app = Flask(\_\_name\_\_)

@app.route('/nlp\_query', methods=['POST'])

def nlp\_query():

query = request.json.get('query', '') # User's natural language query

doc = nlp(query) # Process query with spaCy

pm25\_threshold = 15 # Default PM2.5 threshold

branch\_threshold = 5 # Default branch count threshold

# Parse the query for numerical values

for ent in doc.ents:

if ent.label\_ == 'CARDINAL':

if 'pm2.5' in query.lower():

pm25\_threshold = int(ent.text)

elif 'branches' in query.lower():

branch\_threshold = int(ent.text)

# Construct the SQL query

sql\_query = f"""

SELECT census\_tract, branch\_count, pm25

FROM census\_data

WHERE pm25 > {pm25\_threshold} AND branch\_count > {branch\_threshold};

"""

# Execute the SQL query

conn = get\_db\_connection()

cursor = conn.cursor()

cursor.execute(sql\_query)

data = cursor.fetchall()

conn.close()

return jsonify(data)

**How it works:**

* **spaCy**: Used to process natural language and identify important entities such as PM2.5 and branch counts.
* **Dynamic SQL**: The API then constructs and runs a dynamic SQL query based on the parsed entities, returning the filtered data.

**Summary of How the APIs Were Created:**

* **API 1** (Data Aggregation): Combines data from different sources (SOD, NCUA, EPA) and computes branch density and air quality categorization.
* **API 2** (Dynamic Querying): Supports dynamic queries based on user input for filtering data based on conditions such as PM2.5 and branch count.
* **API 3** (Machine Learning): Predicts PM2.5 levels based on historical pollution data and branch density using a machine learning model.
* **API 4** (NLP Interface): Uses natural language processing (NLP) to interpret user queries and convert them into SQL queries for fetching data.

By using **Flask** to build the APIs, **spaCy** for natural language processing, and **SQLite/MySQL/PostgreSQL** for database interactions, these APIs can be deployed and serve requests dynamically based on user input.