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 accessibility).
  + **Large Language Models (LLMs)**: Deploy an LLM on AWS SageMaker (AWS secret manager for API hiding) 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 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 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 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.

**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 SQLite relational databases. 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 like pandas, sqlite, 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 (**Normalization and validation of master keys**)**

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

* **Install JDBC Driver** to allow PySpark to write directly to SQLite

**Step 5: Verify Data Integration** (confirm the successful load of data into the SQLite database using Python's swlite library)

**Task 4. API Creation**

Several steps were adopted to build a solution that aggregates data, supports dynamic querying, integrates machine learning for predictions, and processes natural language input.

**1. API for Data Aggregation**

The goal 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 SQLite database.
* **Database Setup**: 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 and tables on the common key, then calculate branch density and categorize air quality based on PM2.5 levels.
* **Census Tract Aggregation**: For each Census Tract, branches were counted, 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.

**Implementation steps:**

* **Dynamic Filtering**: The API parses the query parameters, and based on those parameters, constructs a dynamic SQL query.
* **Flexible Input**: Users can adjust the pm2.5\_treshold and branch\_count\_treshold 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.

**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.

**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.