LOW LEVEL DESIGN FOR THYROID DETECTION PROJECT:

1. Overview

This document provides a detailed low-level design for the Thyroid Prediction Project. The project involves predicting thyroid function (normal, hyperfunction, or subnormal) based on a set of medical attributes using a machine learning model deployed on AWS. The system is designed to handle data ingestion, preprocessing, model training, prediction, and user interaction via a web interface.

2. System Components

2.1 User Interface (UI)

- **Description**: A web interface where users can upload their medical data for prediction.
- **Technology Stack**: HTML, CSS, JavaScript.
- **Interaction**: The UI sends the uploaded data file to the Flask API for processing.

2.2 Flask API

- **Description**: A Flask-based API running in an AWS App Runner environment, responsible for handling user requests, processing data, and returning predictions.
- **Technology Stack**: Python, Flask.
- **Endpoints**:
  - `/upload`: Accepts data files uploaded by users.
  - `/predict`: Processes the uploaded data and returns the prediction results.
- **Interactions**:
  - Receives data from the UI.
  - Fetches the trained model from the S3 bucket.
  - Processes the data and makes predictions.
  - Returns the results to the UI.

2.3 S3 Bucket

- **Description**: Stores the trained machine learning model, which is retrieved by the Flask API for prediction.
- **Technology Stack**: AWS S3.
- **Data Stored**: Serialized machine learning model (e.g., `model.pkl`).

2.4 MongoDB

- **Description**: The database where raw thyroid data is stored before being processed.
- **Technology Stack**: MongoDB Atlas.
- **Collections**:
  - `thyroid_data`: Contains all the raw data related to thyroid function.
- **Interactions**:
  - Data is fetched from MongoDB for the ingestion process.

2.5 Model Development Workflow

- **Description**: The workflow involved in developing and training the machine learning model, including data ingestion, preprocessing, validation, transformation, and training.
- **Technology Stack**: Python, Pandas, Scikit-learn.
- **Components**:
  - **Data Ingestion**: Fetches raw data from MongoDB.
  - **Data Preprocessing**: Cleans and prepares the data for further processing.
  - **Data Validation**: Ensures data integrity and consistency against predefined schemas.
  - **Data Transformation**: Converts data into a suitable format for model training.
  - **Model Training**: Trains the machine learning model and stores it in the S3 bucket.

2.6 AWS App Runner

- **Description**: The service hosting the Flask API, responsible for running the prediction service in a scalable and reliable environment.
- **Technology Stack**: AWS App Runner.

3. Detailed Design

3.1 User Interface (UI)

- **File Upload Form**:
  - **HTML**: The form includes a file input for the user to upload their medical data.
  - **JavaScript**: Validates the input file and sends it to the Flask API using an AJAX call.
  - **CSS**: Styles the form for a user-friendly experience.

3.2 Flask API

3.2.1 `upload` Endpoint

- **Method**: `POST`
- **Functionality**:
  - Receives the file from the user.
  - Stores the file temporarily in the server.

    - Calls the `predict` endpoint for processing.

3.2.2 `predict` Endpoint

- **Method**: `POST`
- **Functionality**:
    - Loads the model from the S3 bucket using the `boto3` library.
    - Preprocesses the uploaded data.
    - Applies the model to the data to generate predictions.
    - Attaches the prediction results to the original data.
    - Sends the processed file back to the user for download.

3.3 S3 Bucket

- **Bucket Structure**:
    - **Folder**: `models/`
    - **File**: `thyroid_model.pkl` (Serialized model file)
- **Interaction**:
    - The Flask API retrieves this model using the `boto3` library whenever a prediction is requested.

3.4 MongoDB

3.4.1 Data Ingestion

- **Functionality**:
    - Connects to the MongoDB Atlas instance.
    - Fetches data from the `thyroid_data` collection.
    - Returns the raw data for preprocessing.

3.5 Model Development Workflow

3.5.1 Data Ingestion

- **Code Module**: `data_ingestion.py`
- **Functionality**:
    - Connects to MongoDB and fetches raw data.
    - Saves the data locally for preprocessing.

3.5.2 Data Preprocessing

- **Code Module**: `data_preprocessing.py`
- **Functionality**:
    - Cleans data (handles missing values, outliers).
    - Converts categorical data into numerical formats using one-hot encoding.

3.5.3 Data Validation

- **Code Module**: `data_validation.py`
- **Functionality**:
  - Validates the data against predefined schemas (e.g., `train_schema.json`).
  - Ensures data types, value ranges, and mandatory fields are correct.

3.5.4 Data Transformation

- **Code Module**: `data_transformation.py`
- **Functionality**:
  - Scales numerical features.
  - Applies feature engineering techniques.

3.5.5 Model Training

- **Code Module**: `model_training.py`
- **Functionality**:
  - Splits the data into training and testing sets.
  - Trains the logistic regression model.
  - Performs hyperparameter tuning using cross-validation.
  - Serializes the trained model using `joblib` or `pickle`.
  - Uploads the serialized model to the S3 bucket.

3.6 AWS App Runner

- **Deployment Process**:
  - The Flask application is containerized using Docker.
  - The Docker image is pushed to Amazon ECR (Elastic Container Registry).
  - AWS App Runner is configured to run the containerized Flask API.
  - The environment variables for AWS credentials and S3 bucket details are configured in the App Runner environment.

4. Data Flow

4.1 Prediction Workflow

1. **User uploads data**: The user uploads a data file through the web interface.
2. **File sent to Flask API**: The UI sends the file to the Flask API via the `upload` endpoint.
3. **Model retrieval from S3**: The Flask API retrieves the trained model from S3.
4. **Data Preprocessing**: The Flask API preprocesses the uploaded data to match the format expected by the model.
5. **Prediction**: The Flask API applies the model to the preprocessed data to generate predictions.

6. **Return Results**: The results, now with a prediction column, are returned to the user for download.

4.2 Model Training Workflow

1. **Data Ingestion**: The data ingestion module fetches raw data from MongoDB.
2. **Preprocessing**: The raw data is cleaned and prepared for training.
3. **Validation**: The cleaned data is validated against predefined schemas.
4. **Transformation**: The validated data is transformed into the required format for model training.
5. **Training**: The model is trained using the transformed data.
6. **Serialization and Storage**: The trained model is serialized and uploaded to the S3 bucket for future use.

5. Security Considerations

5.1 Data Privacy

- **User Data**: Ensure that user-uploaded data is securely transmitted and not stored permanently on the server.

5.2 AWS Credentials

- **Environment Variables**: AWS credentials should be stored securely using environment variables in AWS App Runner.
- **IAM Role**: Use an IAM role with limited permissions to access the S3 bucket.

5.3 MongoDB Security

- **Encryption**: Ensure that data in MongoDB is encrypted both at rest and in transit.

6. Error Handling

6.1 Flask API Errors

- **CustomException Handling**: Implement a `CustomException` class to handle errors gracefully within the Flask API. Log errors and return user-friendly messages.

6.2 S3 Access Errors

- **Credential Issues**: Handle errors related to missing or incorrect AWS credentials when accessing the S3 bucket.

6.3 Data Validation Errors

- **Schema Mismatches**: Ensure that any data that fails validation is logged and appropriate error messages are returned to the user.

.