IntelliInspect

Project work done by:

Gedela Sailaja (CS22B1013) | Koppisetty Venkata Sai Siddharth (CS22B1022)
Panchananam Lakshmi Srinivas (CS22B1040) | Kaustub Pavagada (CS22B1042)

Objective:

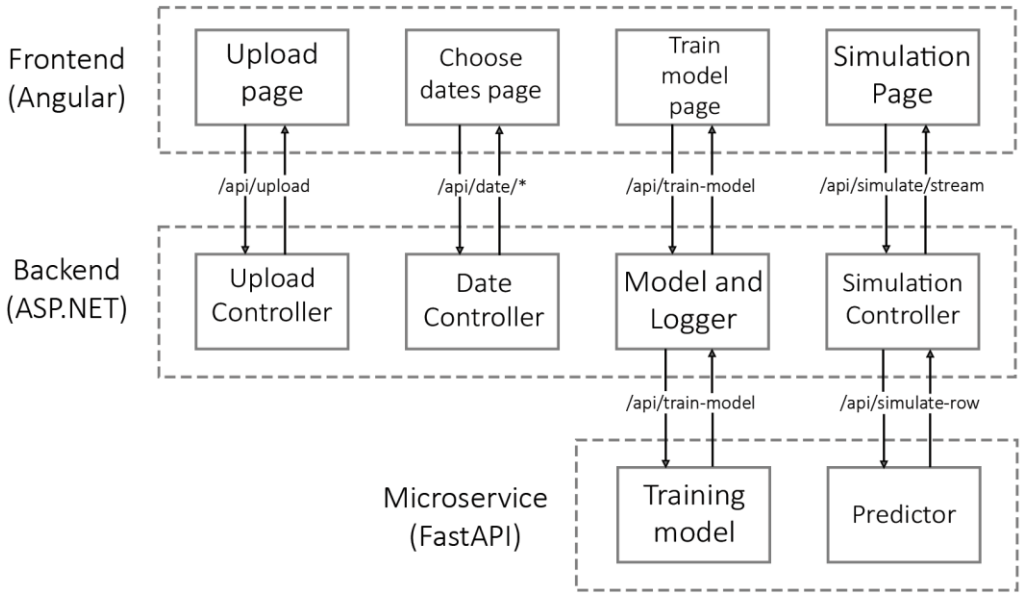
To design and implement a full-stack AI-powered system designed to enable real-time quality control predictions using Kaggle Production Line sensor data. Users can upload CSV files, select a date range, train a machine learning model, and simulate predictions row by row through a simple web interface. The frontend, backend, and ML microservice are fully decoupled and Dockerized for flexibility and scalability. During simulation, each row is sent to the trained model, and predictions are streamed back live to the user, making it ideal for fast, interactive analysis in manufacturing environments.

Sample dataset link: Link

Technologies used:

- Frontend (UI) - **AngularJS**: for the entire user interface and statistics display
- Backend - **ASP.NET Core**: handles all core logic, API routing, and data orchestration between the frontend and the ML microservice. It manages file uploads, date range validations, model training requests, and streaming of row-wise data for real-time predictions.
- Microservice - **FastAPI**: contains the model training code, stores the trained model and returns predictions one-by-one during streaming

System Architecture:



There are three decoupled parts to the implementation. Above shown is the architecture used with the API routes that relate to that particular part of the system.

This type of design enables all services to be independent of each other, at the same time enabling effective communication between them.

Each of the parts are Dockerized inside their respective containers, enabling modular development, scalability, and easier maintenance.

Frontend (Angular):
The Angular frontend consists of multiple pages — Upload, Choose Dates, Train Model, and Simulation — that interact with the backend via HTTP API calls. Each page is responsible for a specific part of the workflow and communicates only with its respective endpoint, promoting a clean separation of concerns in the UI layer.

Backend (ASP.NET):
The backend serves as the orchestration layer. It handles incoming API requests from the frontend, processes them where necessary, and delegates compute-heavy or ML-specific tasks to the microservice. Each controller is mapped to a particular function:

- UploadController: Receives uploaded CSV data and stores it appropriately.
- DateController: Handles all logic related to filtering or choosing data ranges.
- ModelAndLoggerController: Manages model training requests and logs metadata.
- SimulationController: Streams predictions from the trained model in real time.

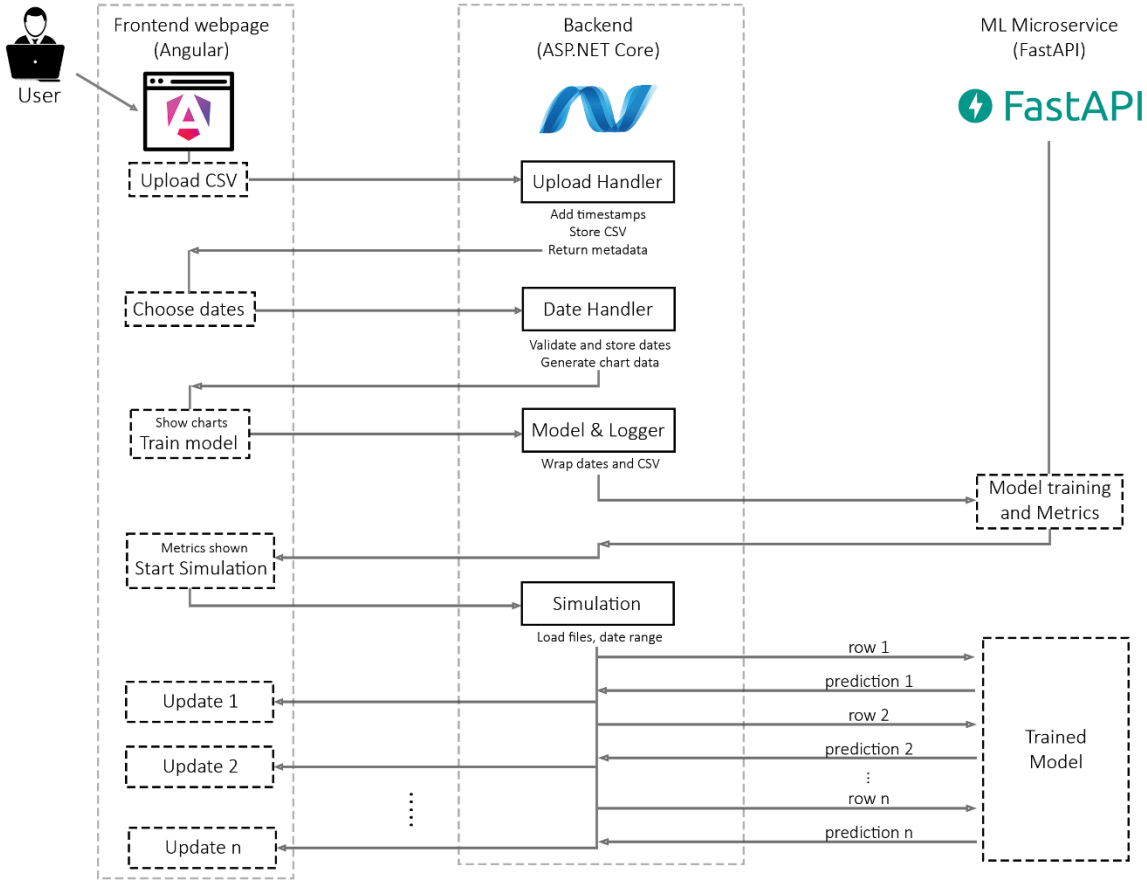
Microservice (FastAPI):
The FastAPI microservice is responsible for the core ML functionality:

- The /train-model endpoint receives preprocessed data and returns a trained model artifact or status.
- The /simulate-row endpoint accepts a single row of data and returns predictions. This service is lightweight, stateless, and can be horizontally scaled for high throughput inference.

This architecture supports:

- **Loose Coupling**: Changes in one service do not affect others.
- **Scalability**: Components like the FastAPI service can scale independently.
- **Flexibility**: Technology stacks are mixed (Angular, .NET, Python) to suit specific layers of the system.
- **Maintainability**: Clear separation of frontend, backend, and ML logic improves readability and debugging.

Data Flow:



The figure above illustrates the **data flow** across the three decoupled parts of the system, emphasizing the user interaction, backend logic, and machine learning pipeline integration during the simulation workflow.

Each interaction proceeds as follows:

- **1. Upload CSV**:
  The user begins by uploading a CSV file through the Angular frontend. This request is handled by the Upload Handler in the ASP.NET backend, which:
  - Adds timestamps (if needed),
  - Stores the uploaded file,
  - Extracts and returns metadata back to the frontend.
- **2. Choose Dates**:
  The user selects a date range, which is sent to the Date Handler. This component:
  - Validates the selected dates,
  - Stores them for later use,
  - Generates relevant chart data for display on the frontend.

- **3. Train Model**:
  When the user initiates model training:
    - The Model & Logger component wraps the CSV and the selected dates,
    - Sends them to the FastAPI microservice endpoint for training,
    - The FastAPI service performs the model training, computes metrics, and returns them.

These metrics are then displayed on the frontend via charts.

- **4. Start Simulation**:
  Once training is complete, the user can begin a simulation. The Simulation component:
    - Loads the stored CSV and the date range,
    - Sends one row at a time to the ML microservice.
- **5. Real-time Prediction**:
  For each row:
    - A request is sent to the FastAPI service using the trained model,
    - The predicted result is returned,
    - The frontend receives and displays the prediction update (e.g., Update 1, Update 2, ..., Update n) in real-time or near real-time.

API Contract and Payload Structure:

1. **POST /api/upload**
   Uploads a CSV file.
   - Request:
     - Content type: multipart/form-data
     - Body: the CSV file to be uploaded
   - Response:
     - 200 OK:
       ```
       {
         "totalRows": int,
         "totalCols": int,
         "columnNames": [ "col1", "col2", ... ],
         "startTimestamp": "YYYY-MM-DDTHH:MM:SS",
         "endTimestamp": "YYYY-MM-DDTHH:MM:SS"
       }
       ```
     - 400 Bad Request (if file is invalid)

2. **GET /api/daterange/constraints**
   Get min/max date constraints from uploaded dataset.
   - Request: None
   - Response:
     - 200 OK:
       ```
       {
         "minDate": "YYYY-MM-DDTHH:MM:SS",
         "maxDate": "YYYY-MM-DDTHH:MM:SS",
       }
       ```
     - 400 Bad Request (if no dataset uploaded)

3. **POST /api/daterange/validate**
   Validate date ranges for training/testing/simulation.
   - Request:
     - Content type: application/json
     - Body:
       ```
       {
         "TrainStart": "YYYY-MM-DDTHH:MM:SS",
         "TrainEnd": "YYYY-MM-DDTHH:MM:SS",
         "TestStart": "YYYY-MM-DDTHH:MM:SS",
         "TestEnd": "YYYY-MM-DDTHH:MM:SS",
         "SimStart": "YYYY-MM-DDTHH:MM:SS",
         "SimEnd": "YYYY-MM-DDTHH:MM:SS"
       }
       ```
   - Response:
     - If valid:
       ```
       {
         "status": "valid",
         "message": "Date ranges validated successfully.",
         "chartData": { ... }
       }
       ```
     - If invalid:
       ```
       {
         "status": "invalid",
         "message": "Date ranges are invalid.",
         "chartData": [ "error1", "error2", ... ]
       }
       ```

4. **POST /api/daterange/submit**
   Save date changes.
   - Request:
     - Same as /api/daterange/validate

- Response:
  - Body:

```
{
  "message": "Date ranges submitted successfully."
}
```

5. **POST /api/train-model**

   Log model training parameters and forward to ML service.

   - Request:
     - Content type: application/json
     - Body: the CSV file to be uploaded
   - Response:
     - 200 OK:

```
{
  "TrainStart": "YYYY-MM-DDTHH:MM:SS",
  "TrainEnd": "YYYY-MM-DDTHH:MM:SS",
  "TestStart": "YYYY-MM-DDTHH:MM:SS",
  "TestEnd": "YYYY-MM-DDTHH:MM:SS",
  "SimulationStart": "YYYY-MM-DDTHH:MM:SS",
  "SimulationEnd": "YYYY-MM-DDTHH:MM:SS"
}
```

     - 404 Not Found (if CSV is missing)

6. **GET /api/Simulation/stream**

   Streams live predictions one-by-one for rows in the uploaded CSV file that fall within the specified simulation date range using Server-Sent Events (SSE).

   - Request:
     - No body
   - Response:
     - Content type: test/event-stream

```
{
  "id": number,
  "timestamp": "YYYY-MM-DDTHH:MM:SS",
  "prediction": binary (0/1),
  "confidence": number,
}
```

     - If CSV not present:

```
{
  "error": "CSV file not found"
}
```

     - If FastAPI returns error:

```
{
  "error": "ML backend error",
  "detail": "<detailed message from microservice"
```

```
}
```