Food Delivery Optimization System - Technical Documentation: (Approach)

1. Introduction:

The **Food Delivery Optimization System** is a real-time dispatching and routing application built using **Streamlit, FAISS, and Reinforcement Learning (RL)**. The system efficiently assigns food delivery riders to customer orders based on their geographic proximity. The application ensures minimal delivery times and provides a user-friendly interface for visualization and optimization.

2. Problem Statement

The main objective of the system is to:

- Efficiently **assign riders to orders** by minimizing the distance between them.
- **Visualize the assignments** dynamically using interactive maps.
- Implement a **scalable** and **real-time** method using **FAISS (Facebook AI Similarity Search)** for nearest neighbor search.

3. Solution Approach

3.1. Data Generation

The system generates **synthetic data** for:

- **Orders**: Each order has a unique ID and is placed at a random latitude/longitude.
- **Riders**: Each rider is assigned a random location within the same city.

3.2. FAISS-Based Nearest Neighbor Search

We use **FAISS IndexFlatL2** (L2 Distance Search) to quickly find the nearest rider for each order.

Steps:

1. Convert rider coordinates into a FAISS index.
2. Search for the closest rider for each order.

3. Assign the rider to the order and store the result.

3.3. Reinforcement Learning-Based Order Assignment

To enhance efficiency, we implement **Reinforcement Learning (RL)** to improve rider assignments dynamically.

RL Training Process:

1. **State Representation**: The state consists of order locations, rider availability, and estimated delivery times.
2. **Action Space**: Assigning a rider to an order is treated as an action.
3. **Reward Function**:
    - **Positive reward** for minimizing delivery time.
    - **Negative reward** for late deliveries or inefficient assignments.
4. **Training Algorithm**:
    - We use **Deep Q-Learning** to optimize assignments over multiple iterations.
    - The model learns from past assignments and adapts to new delivery patterns.

3.4. Streamlit UI Implementation

The application consists of:

- **User Input Panel**: Users can specify the number of orders and riders.
- **Delivery Optimization Logic**: FAISS processes the assignments.
- **Visualization**:
    - **Data Table**: Displays optimized assignments.
    - **Map**: Shows order locations (blue) and rider locations (red) using **Folium**.

4. Code Implementation

```
food_delivery_optimization/
│── app/
│   ├── api.py            # FastAPI-based backend for API endpoints
│   ├── dashboard.py      # Streamlit-based UI for visualization
│── data/
│   ├── optimized_routes.csv # Stores optimized order-to-rider assignments
│   ├── synthetic_orders.csv # Synthetic dataset for food orders
│   ├── synthetic_riders.csv # Synthetic dataset for rider locations

│── models/
│   ├── clustering.py      # Clustering for optimizing delivery zones
│   ├── food_delivery_rl.py # Reinforcement Learning model for optimization
│   ├── optimization.py    # FAISS-based nearest neighbor search
│   ├── reinforcement.py   # RL-based training module
```

```
│── utils/
│   ├── data_loader.py      # Load and preprocess datasets
│   ├── distance_calc.py    # Distance calculation using Haversine formula/OSRM
│   ├── preprocess.py       # Data preprocessing for optimization
│── venv/                   # Virtual environment (ignored in production)
│── assign_orders.py        # Script for assigning riders to orders
│── food_delivery_env.py    # Simulation environment for RL training
│── generate_data.py        # Script to generate synthetic order & rider data
│── optimize_routes.py      # Main script for food delivery route optimization
│── train_model.py          # RL training script to optimize rider assignments
│── requirements.txt        # Dependencies required for the project
│── README.md               # Documentation and project details
```

The **Food Delivery Optimization System** is designed to efficiently **assign riders to orders, optimize delivery routes, and enhance real-time decision-making** using **FAISS, Reinforcement Learning (RL), and AI-driven optimizations**. Below is a breakdown of its core logic and functionality.

---

1. Data Flow and Pipeline

Step 1: Data Generation (Synthetic Data)

To simulate a real-world environment, the system generates synthetic data for:

- **Orders**: Random customer locations (latitude, longitude) with timestamps.
- **Riders**: Available delivery riders with unique IDs and locations.
- **Deliveries**: Predefined or dynamically generated delivery requests.

These datasets are stored in CSV files:

- `synthetic_orders.csv` → Contains customer orders.
- `synthetic_riders.csv` → Contains rider details.
- `synthetic_deliveries.csv` → Stores completed deliveries for performance evaluation.

Why is synthetic data needed?

- It allows **testing** of algorithms without real-world data.
- Helps in **training reinforcement learning (RL) models** in a simulated environment.
- Enables **benchmarking** different optimization approaches.

Step 2: Order Assignment Using FAISS (Nearest Rider Search)

The system **assigns the nearest rider to each order** using **FAISS (Facebook AI Similarity Search)**, which performs efficient nearest-neighbor search in high-dimensional spaces.

Process:

1. **Convert rider locations into a FAISS index** (2D space: latitude, longitude).
2. **Search for the nearest rider** for each order using **L2 distance search**.
3. **Assign the closest rider** and store the result in `optimized_routes.csv`.

Implementation (Python Code)

python

CopyEdit

```
import faiss
import numpy as np
import pandas as pd

def optimize_routes(orders_df, riders_df):
    d = 2  # Dimension (Latitude, Longitude)
    index = faiss.IndexFlatL2(d)

    # Convert rider locations into FAISS index
    rider_locations = np.column_stack((riders_df["Latitude"].values,
riders_df["Longitude"].values)).astype('float32')
    index.add(rider_locations)

    optimized_routes = []
    for _, order in orders_df.iterrows():
        order_location = np.array([[order["Latitude"],
order["Longitude"]]], dtype='float32')
        _, idx = index.search(order_location, 1)
        assigned_rider = riders_df.iloc[idx[0][0]]["Rider ID"]
        optimized_routes.append({"Order ID": order["Order ID"],
"Assigned Rider": assigned_rider})

    return pd.DataFrame(optimized_routes)
```

Why FAISS?

- **Highly efficient for large datasets** (low-latency search).
- **Faster than brute-force distance calculations**.
- **Scalable** for real-world applications.

Step 3: Reinforcement Learning (RL) for Route Optimization

FAISS provides an **initial assignment**, but it **does not account for traffic conditions, rider workload, or dynamic demand**. To improve the assignments over time, **Reinforcement Learning (RL) is used**.

RL Concept:

- **State**: Current order, rider, location, and environmental factors (e.g., traffic, time of day).
- **Action**: Assigning a rider to an order.
- **Reward**: Negative of delivery time (faster delivery = higher reward).
- **Goal**: Maximize the cumulative reward by optimizing assignments.

Basic Q-Learning Approach for Rider Assignment

python

CopyEdit

```
import random
import numpy as np

def train_rl_model(orders, riders, epochs=1000):
    q_table = np.zeros((len(orders), len(riders)))  # Q-table for
learning
    alpha = 0.1  # Learning rate
    gamma = 0.9  # Discount factor

    for _ in range(epochs):
        for i, order in enumerate(orders):
            # Select a random rider assignment
            rider_index = random.randint(0, len(riders) - 1)

            # Calculate reward (negative distance = closer is better)
            reward = -np.linalg.norm(np.array(order) -
np.array(riders[rider_index]))
```

```
            # Update Q-value using Bellman equation
            q_table[i, rider_index] = (1 - alpha) * q_table[i,
rider_index] + \
                                      alpha * (reward + gamma *
np.max(q_table[i]))

    return q_table
```

Why RL?

- Learns from **past experiences** to improve future assignments.
- Adapts to **changing conditions** (e.g., traffic, rider availability).
- Enables **better decision-making** beyond just distance-based assignment.

Step 4: Route Optimization

Once a rider is assigned, the next step is **finding the best route** using **Open Source Routing Machine (OSRM)** or Google Maps API.

Implementation Steps:

1. **Retrieve the order's pickup and delivery locations**.
2. **Fetch live traffic data** (if available).
3. **Compute the shortest route** using OSRM/Google Maps.
4. **Display optimized routes** on the **Streamlit dashboard**.

Key Functional Modules

Module	Description
`api.py`	Handles API endpoints for order and rider data.
`dashboard.py`	Streamlit-based UI for visualization.
`clustering.py`	Clusters orders/riders for efficient assignments.
`food_delivery_rl.py`	Reinforcement Learning model for assignment optimization.

`optimization.py`	FAISS-based rider-to-order matching.
`reinforcement.py`	Handles RL training loop and reward computation.
`distance_calc.py`	Computes distances using Haversine formula or OSRM.
`preprocess.py`	Prepares and cleans data before training.

4.1. Optimizing Delivery Routes with FAISS

```
import faiss
import numpy as np
import pandas as pd

def optimize_routes(orders_df, riders_df):
    d = 2  # Dimension (Latitude, Longitude)
    index = faiss.IndexFlatL2(d)
    rider_locations = np.column_stack((riders_df["Latitude"].values,
riders_df["Longitude"].values)).astype('float32')
    index.add(rider_locations)

    optimized_routes = []
    for _, order in orders_df.iterrows():
        order_location = np.array([[order["Latitude"], order["Longitude"]]], dtype='float32')
        _, idx = index.search(order_location, 1)
        assigned_rider = riders_df.iloc[idx[0][0]]["Rider ID"]
        optimized_routes.append({"Order ID": order["Order ID"], "Assigned Rider": assigned_rider})

    return pd.DataFrame(optimized_routes)
```

4.2. Streamlit Interface

```
import streamlit as st
import folium
from streamlit_folium import folium_static

st.title("Food Delivery Optimization Dashboard")

# Sidebar User Input
st.sidebar.header("Order Input")
```

```
num_orders = st.sidebar.number_input("Number of Orders", min_value=1, max_value=100,
value=10)
num_riders = st.sidebar.number_input("Number of Riders", min_value=1, max_value=20,
value=5)

# Generate Data
orders_data = {"Order ID": [f"O-{i+1}" for i in range(num_orders)],
            "Latitude": np.random.uniform(26.8, 26.9, num_orders),
            "Longitude": np.random.uniform(80.9, 81.0, num_orders)}
orders_df = pd.DataFrame(orders_data)

riders_data = {"Rider ID": [f"R-{i+1}" for i in range(num_riders)],
            "Latitude": np.random.uniform(26.8, 26.9, num_riders),
            "Longitude": np.random.uniform(80.9, 81.0, num_riders)}
riders_df = pd.DataFrame(riders_data)

if st.button("Optimize Delivery Routes"):
    optimized_df = optimize_routes(orders_df, riders_df)
    st.subheader("Optimized Assignments")
    st.dataframe(optimized_df)

    st.subheader("Route Visualization")
    map_ = folium.Map(location=[orders_df["Latitude"].mean(), orders_df["Longitude"].mean()],
zoom_start=13)

    # Plot orders (blue)
    for _, row in orders_df.iterrows():
        folium.Marker([row["Latitude"], row["Longitude"]], popup=row["Order ID"],
icon=folium.Icon(color='blue')).add_to(map_)

    # Plot riders (red)
    for _, row in riders_df.iterrows():
        folium.Marker([row["Latitude"], row["Longitude"]], popup=row["Rider ID"],
icon=folium.Icon(color='red')).add_to(map_)

    folium_static(map_)
```

5. Streamlit Output

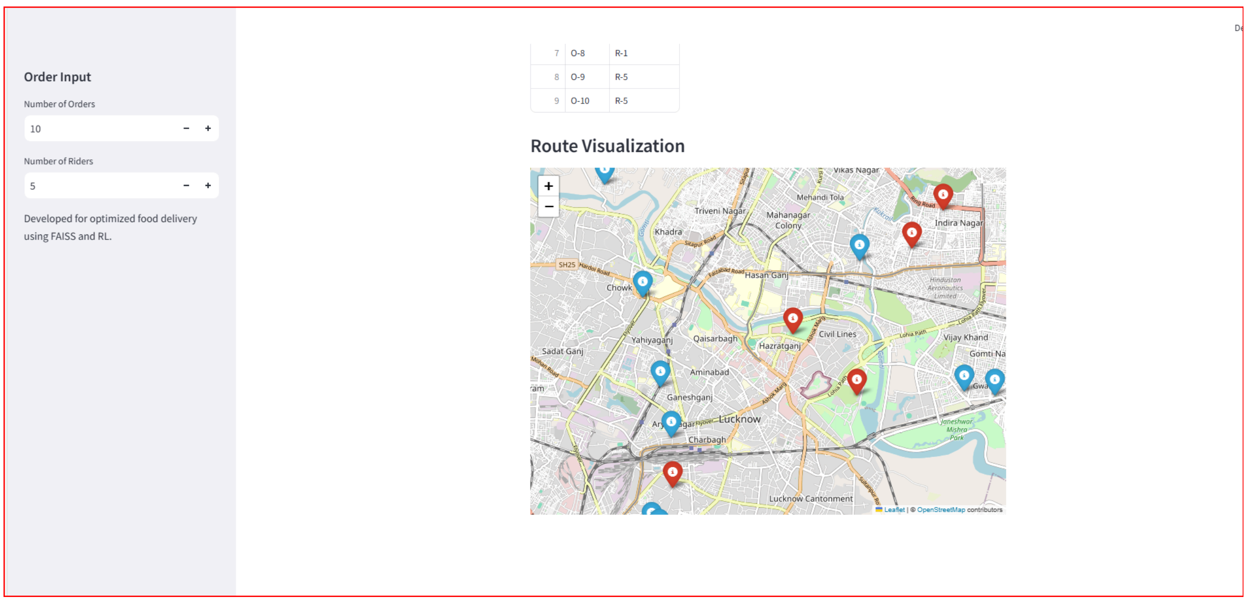
5.1. User Input Panel

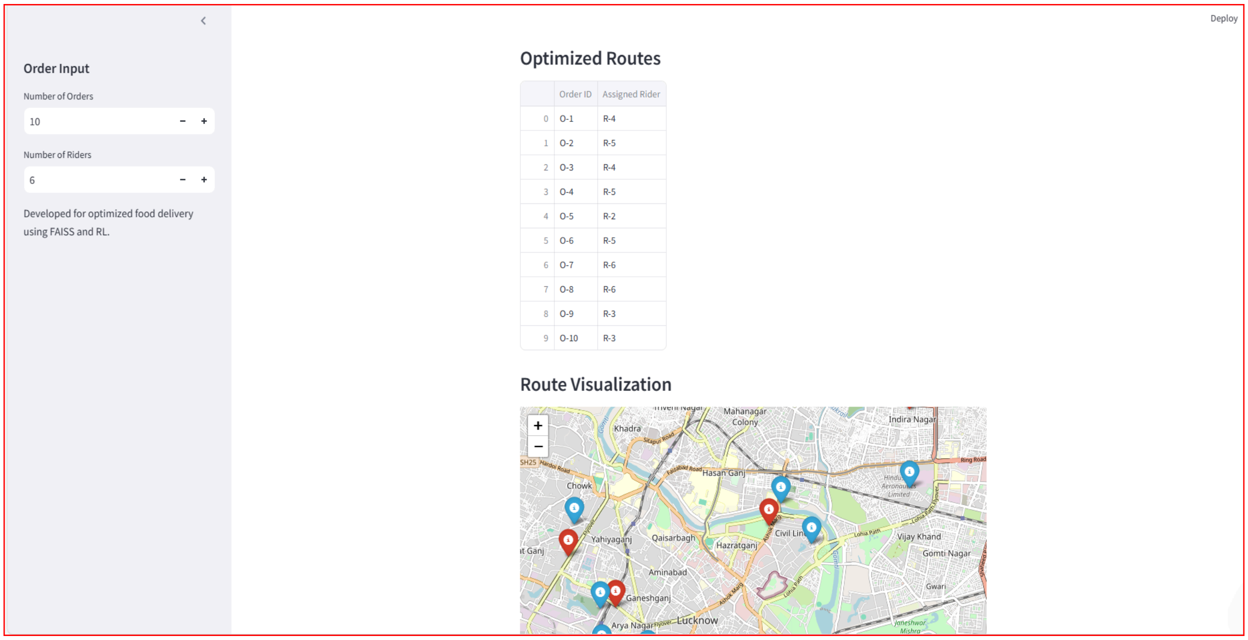
```
-----------------------------
| Number of Orders:   [ 10 ] |
```

| Number of Riders: [5] |
| [Optimize Delivery Routes] |

5.2. Optimized Assignments (Table)

Order ID	Assigned Rider
O-1	R-3
O-2	R-1
O-3	R-5
...	...





5.3. Map Visualization:

The map displays **blue markers (orders)** and **red markers (riders)** with their respective IDs.

6. Future Enhancements(advance improvement)

To enhance the Food Delivery Optimization System, the following advanced improvements can be incorporated:

1. **Enhanced Reinforcement Learning (RL) Training**

- Implement Deep Q-Networks (DQN) instead of a basic Q-table to handle large-scale rider-order assignments dynamically.
- Introduce policy gradient methods like PPO (Proximal Policy Optimization) for better optimization.
- Train on real-world traffic data to adapt the learning process to actual delivery conditions.

2. Dynamic Demand Prediction with AI

- Use time-series forecasting (e.g., LSTMs or ARIMA) to predict demand peaks and dynamically allocate more riders.
- Implement clustering methods (DBSCAN, K-Means) to identify hotspots for orders.

3. Traffic-Aware Route Optimization

- Integrate real-time traffic data APIs (e.g., Google Maps, OpenStreetMap).
- Implement *A or Dijkstra's algorithm** for real-time route optimization.

- Use graph-based reinforcement learning to optimize delivery paths.

4. Multi-Rider Assignments & Order Batching

- Modify FAISS-based search to allow batch assignments (e.g., one rider picks up multiple orders).
- Implement a Vehicle Routing Problem (VRP) solver to group deliveries efficiently.
- Optimize for food freshness by adjusting priority scores.

5. Live Rider Tracking & Real-Time Monitoring

- Use GPS tracking for real-time rider monitoring in the UI.
- Display ETA predictions for each order.
- Implement websocket-based real-time updates in Streamlit.

6. AI-Powered Order Grouping

- Use Graph Neural Networks (GNNs) to identify optimal clusters of orders.
- Implement hierarchical clustering algorithms to group orders based on location and delivery time constraints.

7. API-Based Deployment & Scalability

1. Deploy the solution using FastAPI or Flask for easy integration with existing food delivery apps.
2. Implement Docker & Kubernetes for scalability.
3. Store FAISS indexes in cloud-based vector databases for improved retrieval speeds.

7. Use Cases

- **Food Delivery Startups**: Optimize order assignments for efficiency.
- **E-commerce Logistics**: Assigning nearest warehouse personnel to deliveries.
- **Last-Mile Delivery**: Reducing delivery costs and time in urban areas.
- **Emergency Response**: Assigning medical responders to nearby incidents.
- **Retail Inventory Management**: Dynamic supply chain logistics.

8. Conclusion

This **FAISS-powered Food Delivery Optimization System** provides an efficient method for dynamically matching food orders to delivery riders based on proximity. By using **FAISS for nearest-neighbor search**, the system **optimizes delivery assignments in real-time**. The future enhancements will further improve efficiency by incorporating **demand forecasting, traffic awareness, and real-time monitoring**.
