---

1. Data Ingestion Pipelines

Batch Data Pipeline

- **Purpose**: To process historical data for model training and periodic retraining.
- **Data Sources**:
  - Historical ride data: Pickup/drop-off locations, ride types, timestamps.
  - Historical driver data: Ratings, acceptance rates, ride types.
- **Tools**:
  - **Apache Spark**: For data transformation and cleaning.
  - **AWS S3**: For storage of processed historical data.
- **Flow**:
  1. Extract data from relational databases (e.g., PostgreSQL).
  2. Transform and clean data using Spark jobs.
  3. Load processed data into S3 for downstream use.

Real-Time Data Pipeline

- **Purpose**: To handle live ride requests and driver availability updates.
- **Data Sources**:
  - Ride requests: Pickup/drop-off locations, ride types.
  - Driver updates: Location, availability, ride type preferences.
- **Tools**:
  - **Apache Kafka**: For streaming real-time data.
  - **AWS Lambda**: For triggering downstream pipelines.
- **Flow**:
  1. Ingest ride request and driver update events via Kafka.
  2. Stream data to a processing layer for immediate inference and optimization.

---

2. Data Transformation and Feature Engineering

Feature Engineering

- **Purpose**: To generate meaningful features for the ML model.
- **Steps**:
  - **ETA Calculation**: Compute estimated time to the pickup location for each driver.
  - **Feature Encoding**: Convert categorical variables (e.g., ride type) into numerical representations.
  - **Geofencing**: Assign drivers and ride requests to specific geographic zones to reduce search space.
- **Output**:

- Transformed data is stored in an **AWS Feature Store** for reuse across training and inference pipelines.

---

3. ML Model Training Pipeline

Model Details

- **Algorithm**: Deep Reinforcement Learning (e.g., DQN or PPO).
- **Input Features**:
  - Driver: Location, availability, ride type preference, rating, acceptance rate.
  - Ride Request: Pickup/drop-off locations, ride type.
  - Environmental: Real-time traffic data, geofencing zones.

Pipeline

1. **Data Preparation**:
   - Pull data from AWS Feature Store and S3.
   - Split into training, validation, and test sets.
2. **Model Training**:
   - Train the RL model to predict compatibility scores for driver-ride pairs.
   - Use historical data to simulate driver and passenger behaviors.
3. **Validation & Testing**:
   - Evaluate the model using a separate validation set.
   - Ensure low latency predictions for real-time inference.
4. **Model Registry**:
   - Register the trained model in an AWS Model Registry for deployment.

---

4. Real-Time Inference

Steps

1. Fetch real-time data from Kafka streams.
2. Retrieve precomputed features from the AWS Feature Store.
3. Use the trained RL model to compute scores for all possible driver-ride pairs.
4. Send the scores to the graph-based optimization layer.

---

5. Graph Construction and Optimization

Graph Construction

- **Purpose**: Represent the driver-ride assignment problem as a bipartite graph.
- **Details**:
  - Nodes:
    - Drivers.
    - Ride requests.
  - Edges:
    - Compatibility scores from the ML model.

Graph-Based Optimization

- **Algorithm**: Hungarian Algorithm or Minimum Cost Flow.
- **Goal**:
  - Minimize total ETA.
  - Maximize assignment efficiency.
- **Output**:
  - Optimal driver-ride pairings.

---

6. Ride Assignment

- **Details**:
  - Assign drivers to rides based on optimized pairings.
  - Update driver availability in real-time.
- **Tools**:
  - AWS Lambda: For triggering the ride assignment process.
  - Kafka: For notifying drivers and passengers.

---

7. Retraining Pipeline

Trigger Conditions

- Periodic retraining (e.g., weekly).
- Data drift detection (e.g., significant changes in driver behavior or ride patterns).

Pipeline

1. Monitor data drift using real-time metrics.
2. Store new data in S3 and AWS Feature Store.
3. Trigger the model training pipeline for retraining.
4. Validate and deploy the updated model.

---

8. Geo-Fencing Implementation

- **Purpose**:
  - Improve scalability by dividing the operational area into smaller zones.
  - Restrict driver-ride pairing computations within zones.
- **Process**:
  1. Assign geofencing zones to both drivers and ride requests based on lat/long.
  2. Run graph optimization independently for each zone.
  3. Merge results for global ride assignment.
