Flight Fare Prediction - Detailed Architecture Document

Table of Contents

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the Flight Fare Prediction system. It aims to offer insights into the design decisions, component interactions, and technical specifications that form the backbone of this machine learning-driven prediction system.

1.2 Scope

The architecture described herein encompasses:

- Data Collection and Preprocessing
- Machine Learning Model Implementation (Random Forest)
- Database Design and Management (Cassandra DB)
- API Development and Exposure (Flask/FastAPI)
- Deployment Strategies (Local and Cloud)
- Monitoring and Logging Systems
- Security Measures
- Performance Optimization Techniques

1.3 Definitions, Acronyms, and Abbreviations

- ML: Machine Learning
- RF: Random Forest
- API: Application Programming Interface
- DB: Database
- NoSQL: Not Only SQL
- REST: Representational State Transfer
- JSON: JavaScript Object Notation

1.4 References

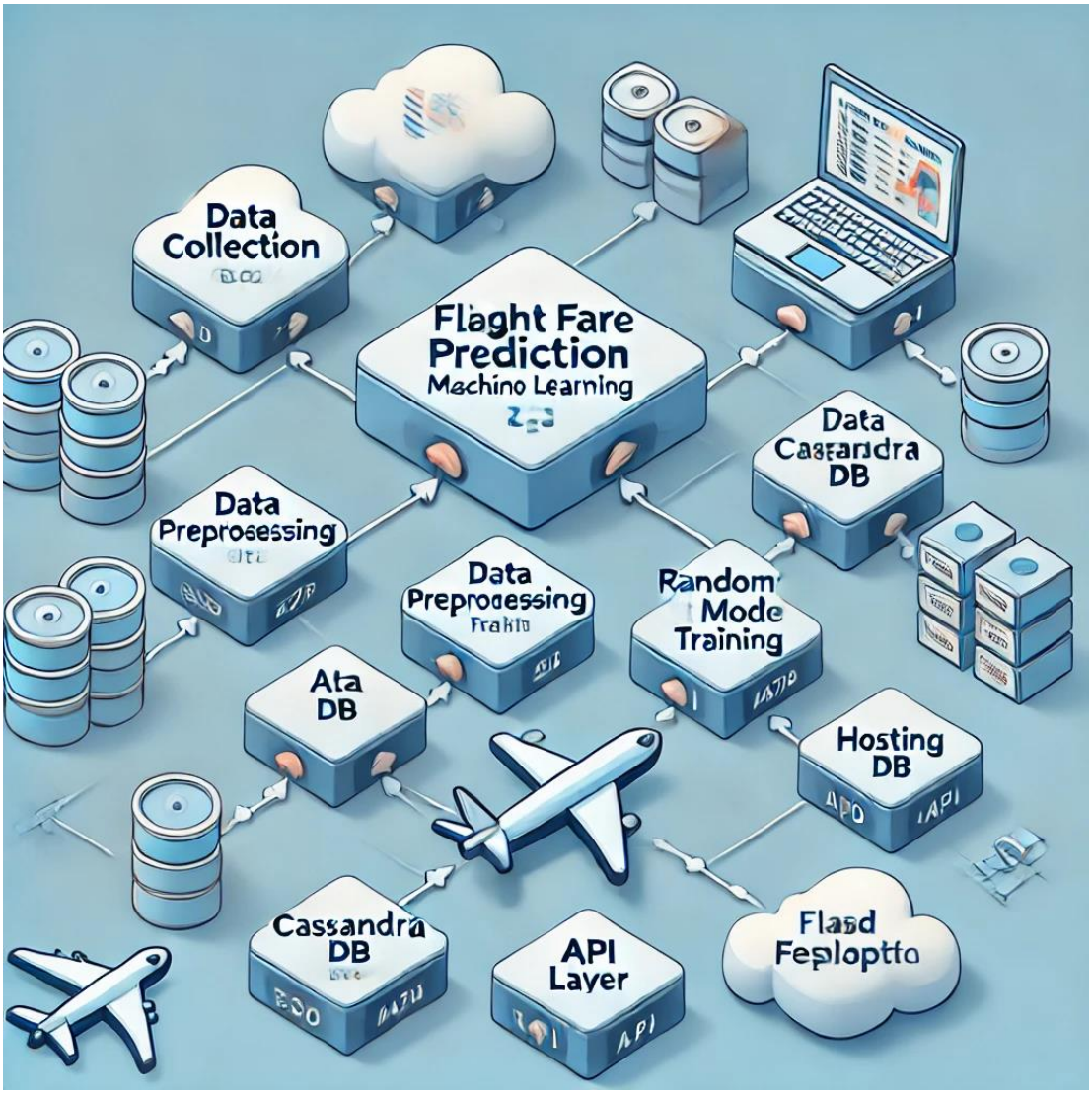
- Scikit-learn Documentation
- Cassandra DB Documentation
- Flask Documentation
- FastAPI Documentation

2. System Overview

2.1 Architectural Representation

High-Level Architecture Diagram

The above diagram illustrates the high-level architecture of the Flight Fare Prediction system. It showcases the interplay between various components and the flow of data through the system.



2.2 System Components

1. Data Collection Module
2. Data Preprocessing Pipeline

3. Random Forest Model
4. Cassandra Database
5. Flask/FastAPI Service
6. Deployment Environment (Local/Cloud)
7. Monitoring and Logging System

2.3 Data Flow

1. Raw flight data is collected from various sources.
2. The preprocessing pipeline cleans and transforms the data.
3. Processed data is used to train the Random Forest model.
4. The trained model is exposed via an API.
5. New data for predictions is received through the API.
6. Predictions are made and returned to the user.
7. Relevant data and predictions are stored in Cassandra DB.
8. The entire process is monitored and logged for performance and security.

3. Data Collection and Preprocessing

3.1 Data Sources

- Historical flight data from airlines
- Real-time flight information APIs
- User-submitted data through the application interface

3.2 Data Collection Process

1. API Integration: Real-time data is fetched using RESTful API calls to partner services.
2. Batch Processing: Historical data is imported in batches using ETL (Extract, Transform, Load) processes.
3. User Input: Direct user input is captured through the application's front-end interface.

3.3 Preprocessing Pipeline

1. Data Cleaning
    - Handling missing values: Implement imputation techniques or remove incomplete records based on the extent of missing data.
    - Removing duplicates: Identify and eliminate duplicate entries to maintain data integrity.
    - Correcting inconsistencies: Standardize data formats, units, and representations across the dataset.
2. Feature Engineering
    - Temporal features: Extract day of week, month, season from flight dates.
    - Categorical encoding: Implement one-hot encoding for categorical variables like airlines, source, and destination.
    - Numerical scaling: Apply standardization or normalization to numerical features like flight duration.
3. Feature Selection
    - Correlation analysis: Identify and remove highly correlated features to reduce multicollinearity.
    - Feature importance: Utilize Random Forest's built-in feature importance to select the most relevant predictors.
4. Data Transformation

  - Log transformation: Apply to fare prices to handle skewness and make the distribution more normal.
  - Binning: Create bins for continuous variables like flight duration or distance if necessary.

3.4 Data Validation

- Implement data validation checks to ensure data quality and consistency.
- Set up automated alerts for anomalies in the incoming data.

4. Machine Learning Model

4.1 Model Selection

The Random Forest algorithm was chosen for its:

- Ability to handle non-linear relationships
- Robustness to outliers and noisy data
- Built-in feature importance ranking
- Ensemble nature, which helps in reducing overfitting

4.2 Random Forest Architecture

Random Forest Architecture

4.3 Model Training Process

1. Data Splitting
    - Training set: 70% of the data
    - Validation set: 15% of the data
    - Test set: 15% of the data
2. Hyperparameter Tuning
    - Use Grid Search with 5-fold cross-validation to optimize:
        - Number of trees (n_estimators)
        - Maximum depth of trees (max_depth)
        - Minimum number of samples required to split an internal node (min_samples_split)
        - Minimum number of samples required to be at a leaf node (min_samples_leaf)
3. Model Training
    - Train the Random Forest model using the optimized hyperparameters on the training dataset.
4. Model Evaluation
    - Evaluate the model on the validation set using metrics such as Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and R-squared.
    - Fine-tune the model if necessary based on the validation results.
5. Final Testing

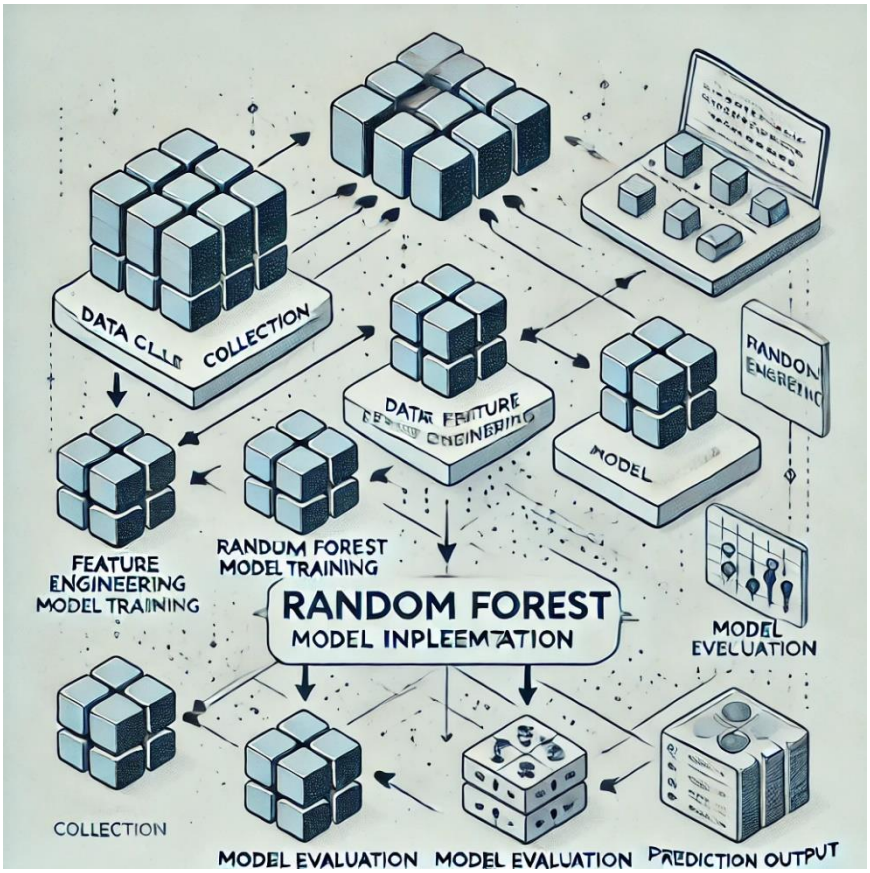
- Assess the final model performance on the held-out test set to get an unbiased estimate of its predictive capability.

4.4 Feature Importance Analysis

- Utilize the Random Forest's built-in feature importance ranking to understand which factors most significantly influence flight fares.
- Visualize feature importances to provide insights to stakeholders.

4.5 Model Persistence

- Save the trained model using joblib or pickle for easy loading during prediction phase.
- Implement versioning for model files to track changes over time.



5. Database Architecture

5.1 Database Selection

Cassandra DB was chosen for its:

- High write throughput
- Linear scalability
- Flexible schema design
- Ability to handle large volumes of data across distributed systems

5.2 Data Model

Flight Data Table

```
CREATE TABLE flight_data (
    flight_id UUID PRIMARY KEY,
    airline TEXT,
    source TEXT,
    destination TEXT,
    departure_time TIMESTAMP,
    arrival_time TIMESTAMP,
    duration INT,
    stops INT,
    fare DECIMAL
);
```

Prediction Log Table

```
CREATE TABLE prediction_log (
    prediction_id UUID PRIMARY KEY,
    flight_id UUID,
    predicted_fare DECIMAL,
    actual_fare DECIMAL,
    prediction_time TIMESTAMP,
    features MAP<TEXT, TEXT>
);
```

5.3 Indexing Strategy

- Create secondary indexes on frequently queried columns like 'airline', 'source', and 'destination'.
- Implement a custom index for date range queries on 'departure_time'.

5.4 Data Consistency

- Use consistency level of QUORUM for writes to ensure data durability.

- Implement read repairs to maintain eventual consistency across all nodes.

5.5 Backup and Recovery

- Schedule regular backups using Cassandra's snapshots feature.
- Implement a recovery plan with clearly defined Recovery Point Objective (RPO) and Recovery Time Objective (RTO).

6. Deployment Architecture

7.1 Local Deployment

- Containerize the application using Docker for consistency across environments.
- Use docker-compose for orchestrating multiple containers (API, database, etc.).

7.2 Cloud Deployment

- Utilize Kubernetes for container orchestration in the cloud.
- Implement auto-scaling based on CPU utilization and request rate.

7.3 Continuous Integration/Continuous Deployment (CI/CD)

- Use GitLab CI/CD pipelines for automated testing and deployment.
- Implement blue-green deployment strategy for zero-downtime updates.

7.4 Environment Management

- Use environment variables for configuration management.
- Implement separate configurations for development, staging, and production environments.

8. Monitoring and Logging

8.1 Logging Strategy

- Utilize structured logging with JSON format for easy parsing.
- Implement log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) for granular control.

8.2 Monitoring Metrics

- API performance: Request rate, response times, error rates
- Model performance: Prediction accuracy, feature importance drift
- System health: CPU usage, memory consumption, disk I/O

8.3 Alerting

- Set up alerts for critical issues like high error rates or system resource exhaustion.
- Use PagerDuty for on-call rotations and incident management.

8.4 Visualization

- Implement Grafana dashboards for real-time monitoring of system metrics.
- Use Kibana for log analysis and visualization.

9. Security Considerations

9.1 Data Encryption

- Implement TLS for all data in transit.
- Use AES-256 encryption for sensitive data at rest.

9.2 Authentication and Authorization

- Implement JWT-based authentication for API access.
- Use role-based access control (RBAC) for fine-grained permissions.

9.3 Input Validation

- Validate all user inputs on the server-side to prevent injection attacks.
- Implement strict type checking and range validation for numerical inputs.

9.4 Dependency Management

- Regularly update dependencies to patch known vulnerabilities.
- Use tools like Snyk for continuous vulnerability scanning.

10. Performance Optimization

10.1 Caching Strategy

- Implement Redis for caching frequent predictions and API responses.
- Use cache invalidation strategies to ensure data freshness.

10.2 Database Optimization

- Implement database connection pooling to reduce connection overhead.
- Use batch processing for bulk inserts and updates.

10.3 API Optimization

- Implement pagination for endpoints returning large datasets.
- Use asynchronous programming to handle concurrent requests efficiently.

11. Future Enhancements

11.1 Advanced ML Models

- Explore ensemble methods combining Random Forest with other algorithms like XGBoost.
- Investigate deep learning approaches for capturing complex patterns in flight pricing.

11.2 Real-time Data Integration

- Implement streaming data processing using Apache Kafka for real-time updates to the prediction model.

11.3 Explainable AI

- Integrate SHAP (SHapley Additive exPlanations) values for more interpretable predictions.

11.4 A/B Testing Framework

- Develop an A/B testing framework to compare different model versions in production.

12. Conclusion

This detailed architecture document provides a comprehensive overview of the Flight Fare Prediction system. It covers all aspects of the system from data collection to deployment and future enhancements. The modular and scalable design ensures that the system can evolve to meet changing requirements and incorporate new technologies as they emerge.

By following this architecture, the development team can build a robust, efficient, and maintainable system for predicting flight fares, providing value to both the business and end-users.