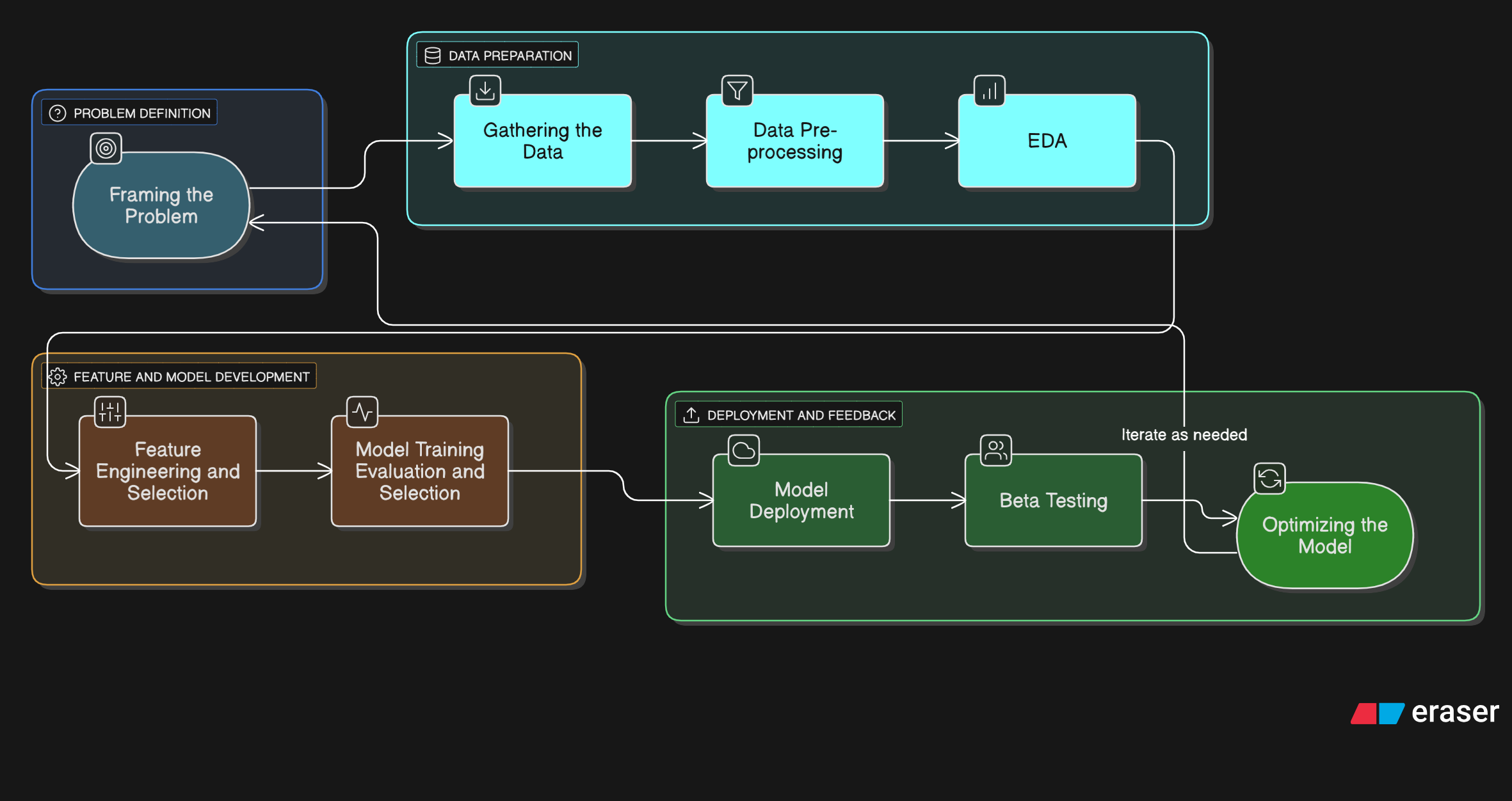
**Car Price Project Details**

The Car Price Prediction project studies the quikr\_car dataset to learn patterns and relationships between car attributes (e.g., name, company, year, kms\_driven, fuel\_type) and price. The primary objective is modeling and inference: build and validate models that can accurately predict the price for resale, unseen input records. This project emphasizes (a) careful data investigation and feature engineering, (b) robust validation so the model generalizes to new inputs, and (c) a production-ready inference endpoint for serving predictions.

Project Life-Cycle –



1. **Framing the Problem: -** This project aims to predict the selling price of used cars using their key attributes, including company name, model name, year of manufacture, kilometers driven, and fuel type. The target variable is the car’s price, which is continuous in nature. Accurate prediction is important for ensuring fair valuation in the used car market, helping both buyers and sellers make informed decisions. The problem is framed as a **supervised machine learning regression task**, where the goal is to model the relationship between car features and their selling price.

To achieve this, multiple regression algorithms are applied, including **Linear Regression** for a baseline, and **Random Forest** and **XGBoost** for more complex, non-linear modeling. These algorithms are chosen to balance **interpretability and predictive power**, with ensemble methods like Random Forest and XGBoost expected to capture feature interactions more effectively. Model performance will be evaluated using the **R² score**, complemented by error metrics such as **MAE** or **RMSE**, to ensure accurate prediction and generalization to unseen car data.

1. **Gathering the data**: - The data was scraped from Quikr.com ([https://quikr.com](https://quikr.com/))

For this project, essential Python libraries were imported:

pandas for data manipulation and analysis

numpy for numerical operations

matplotlib.pyplot for visualization

The dataset containing used car listings was loaded from a local CSV file. To ensure the code works even if the project is uploaded to GitHub, a fallback URL option is included:

import os

car = pd.read\_csv('quikr\_car.csv') if os.path.exists("quikr\_car.csv") else pd.read\_csv("https://raw.githubusercontent.com/username/repo/main/quikr\_car.csv")

The dataset has rows representing car listings and columns representing features like name, company, year, Price, kms\_driven, and fuel\_type.

Using car.shape and car.info(), we observed:

* Some numeric columns (year, Price, kms\_driven) are stored as objects.
* Price contains non-numeric entries such as "Ask For Price" and includes commas.
* kms\_driven has text like " km" appended to numeric values.
* year has non-numeric entries.
* Missing values exist in fuel\_type.
* name values are long and inconsistent, requiring standardization.

These observations indicated the need for **data cleaning and preprocessing** before analysis or modelling.

1. **Data Pre-processing**: - The raw dataset contained inconsistencies and noisy values that could affect model performance. A systematic cleaning and transformation process was performed as follows:

Handling year Column

Observed that year column had some non-year values and was stored as an object.

Converted the column to numeric by keeping only rows where the values were valid numbers:

car = car[car['year'].str.isnumeric()]

car['year'] = car['year'].astype(int)

This removed rows with invalid year entries and ensured year could be used for numerical calculations.

Cleaning Price Column

The Price column contained a non-numeric value "Ask For Price" and commas in numeric entries.

Excluded rows with "Ask For Price" and removed commas, then converted the column to integer type:

car = car[car['Price'] != 'Ask For Price']

car['Price'] = car['Price'].str.replace(',', '').astype(int)

Identified outliers: the median price (~300,000) and mean (~411,717) indicated a few very high values.

Any car with Price > 6,000,000 was treated as an outlier and removed.

Cleaning kms\_driven Column

Values contained ' kms' text and some missing values (NaN).

Extracted numeric portion and removed non-numeric rows:

car['kms\_driven'] = car['kms\_driven'].str.split(' ').str.get(0).str.replace(',', '')

car = car[car['kms\_driven'].str.isnumeric()]

car['kms\_driven'] = car['kms\_driven'].astype(int)

Rows with missing or non-numeric kms\_driven were removed to ensure all entries are valid integers.

Handling fuel\_type Column

fuel\_type contained some missing values (NaN).

Rows with missing fuel\_type were removed to ensure consistency:

car = car[car['fuel\_type'].notna()]

Cleaning name Column

Car names were often long and inconsistent, which could affect analysis.

Reduced each car name to the first three words to standardize naming:

car['name'] = car['name'].str.split(' ').str.slice(0, 3).str.join(' ')

Final Steps

Reset the DataFrame index after removing rows:

car = car.reset\_index(drop=True)

Verified data types: year, Price, and kms\_driven are now integers; fuel\_type and name are strings.

Saved the cleaned dataset for further analysis:

car.to\_csv('Cleaned\_Car\_data.csv', index=False)

1. **Exploratory Data Analysis Explanation (EDA)**: - The goal of this project is to predict used car prices. EDA was performed to understand the relationship between the target variable (Price) and key features (company, year, kms\_driven, fuel\_type).
   1. **Relationship of Company with Price: -** The **boxplot** interpreting relationship between car company and price.

Variation in Price by Company

* + Some companies (like Audi, Mercedes, Land Rover, Jaguar, Volvo) have higher median prices compared to others.
  + Companies like Hyundai, Maruti, Skoda, Datsun, Tata have lower median prices (more affordable segment).

Luxury Brands vs. Budget Brands

* + Luxury car makers (Audi, BMW, Mercedes, Jaguar, Volvo, Land Rover) show significantly higher price ranges, with medians often above ₹1 million (10 lakh).
  + Mass-market brands (Hyundai, Maruti, Tata, Honda, Renault, Ford, Skoda) are clustered in the lower range (< ₹0.5 – 0.7 million).

Spread (Price Range)

* + Companies like Toyota, Audi, Mercedes have a very wide interquartile range (IQR) → they sell both mid-range and high-end cars.
  + In contrast, brands like Datsun, Fiat, Mini show narrower spreads, suggesting fewer models and a more consistent pricing segment.

Outliers

* + Almost every brand has outliers (dots outside the whiskers), representing particularly cheap or expensive cars relative to their main product line.
  + Example: Hyundai and Honda have some higher-priced outliers, likely premium models compared to their usual budget offerings.

Clusters of Affordability

* + Low-end segment: Hyundai, Maruti, Tata, Datsun, Skoda.
  + Mid-range: Honda, Renault, Ford, Nissan.
  + High-end/luxury: Audi, BMW, Mercedes, Jaguar, Volvo, Land Rover.

Conclusion:

* Budget-oriented brands (Hyundai, Maruti, Tata, Datsun) dominate the lower price range.
* Luxury brands (Mercedes, Audi, Jaguar, Land Rover, Volvo) dominate the premium/high price segment.
* Some brands like Toyota and BMW span across multiple segments, offering both affordable and premium cars.
* Outliers indicate that each company may have at least one model priced very differently from the rest (e.g., a luxury version or a budget version in a mostly premium company).
  1. **Relationship of Year with Price: -**- The **scatterplot** interpreting price changes over the years.

Older Cars (before ~2005):

* + Prices are much lower (clustered below ₹0.5M).
  + A few rare outliers (classic cars, luxury imports) appear at higher prices.

2005–2010 Range:

* + Prices start spreading upward, but still mostly under ₹1M.
  + Indicates more availability of mid-range models.

2010–2015 Range:

* + Big jump in car prices, with many cars between ₹0.5M – ₹1.5M.
  + Luxury cars become more common in the dataset.

2015 onward:

* + Prices show the widest spread:
    - Budget cars still exist (~₹0.2M).
    - But a huge number of premium cars above ₹2M also appear.
  + Outliers around ₹3M → high-end luxury cars (Jaguar, Mercedes, Land Rover, etc.).

Conclusion:

* There is a clear upward trend in prices over time.
* Older cars (<2005) are cheaper (likely due to depreciation + fewer luxury options).
* After ~2010, prices diversify: you see both budget and high-end luxury cars.
* Recent years show a wider gap between economy and luxury segments.
  1. **Relationship of kms\_driven with Price:** The **relplot** interpreting price with kms\_driven-

Negative relationship (general trend):

* + As kms\_driven increases, car Price tends to go down.
  + Cars with low mileage (<50k km) have a wide range of prices, from cheap to very expensive.
  + Cars with high mileage (>100k km) are mostly lower-priced.

Clustered data:

* + Most cars are between 0–100k km and priced between 0–1,000,000 (10 lakh).
  + This makes sense because most used cars fall in that typical range.

Outliers:

* + Some cars with very high prices (~2M–3M = 20–30 lakh) even though mileage is low — likely luxury cars (BMW, Audi, Mercedes).
  + A few with high mileage but still priced high — might be rare models or data entry issues.

Conclusion:

* Confirms business logic: more driven cars → less valuable.
* Helps identify outliers you may want to remove before training ML model.
* Shows non-linear trend (not perfectly straight line).
  1. **Relationship of Fuel Type with Price:** The **boxplot** interpreting Price with Fuel Type-
* Diesel cars generally have higher median prices than petrol; CNG/LPG are cheapest.
* Outliers often represent luxury brands, which are expensive regardless of fuel type.
* Conclusion: Fuel type significantly affects price and should be included in the model.
  1. **Relationship of Price with FuelType, Year and Company mixed**

Price Variation by Company

* + Premium brands (e.g., BMW, Mercedes) likely have higher prices.
  + Mass-market brands (e.g., Hyundai, Ford) cluster at lower price ranges.

Fuel Type Patterns

* + Diesel cars may have slightly higher prices than petrol within the same company.
  + Electric or hybrid cars (if present) often appear as highest-priced points, sometimes clustered at newer years.

Year Effects

* + Larger points represent newer cars.
  + Newer cars tend to be more expensive, showing upward trend in the scatter.
  + Older cars (small points) often appear at lower prices, regardless of company.

Mixed Trends

* + Within a single company, price often increases with newer year.
  + Across companies, the baseline price differs: e.g., BMW vs Ford for the same year.
  + Fuel type may slightly shift price for a given company-year combination.

Conclusions:

Price is influenced by multiple factors:

* + Company: Primary driver of baseline price.
  + Year: Newer cars are generally more expensive.
  + Fuel type: Diesel/electric slightly increase price; petrol often cheaper.

Interactions exist:

* + Premium brands + newer year + diesel/electric → highest prices.
  + Economy brands + older year + petrol → lowest prices.

Insights for Pricing Analysis

* + To predict car prices accurately, you cannot rely on a single factor; you need a model combining company, fuel type, and year.
  + Visualizing size by year clearly shows age’s impact on price, which might be hidden in simple boxplots.

Price is influenced by multiple factors: company, year, fuel type, and mileage.

Outliers exist across brands and should be treated carefully.

**Linear Regression** is suitable for this dataset: Fast and interpretable for small-to-medium datasets. Coefficients show expected price changes per unit change in year or relative to baseline company/fuel type. Captures main trends such as: newer cars → higher price, premium companies → higher price, diesel → slightly higher price.

**Random Forest Regression** is suitable for this dataset: Less sensitive to outliers than LR. Doesn’t require scaling of features. Works well with both categorical and numerical features (after encoding)

Also, we will apply XGBoost (XGB), Can capture subtle patterns in combinations of features like company + fuel\_type + kms\_driven.

1. **Feature Engineering and Selection: -**

Feature Engineering: - To improve model performance and better capture relationships between car attributes and price, several new features were created:

Car Age (car\_age): Calculated as the difference between the current year (2025) and the car’s manufacturing year:

car['car\_age'] = 2025 - car['year']

This captures the natural depreciation of a vehicle over time, which is highly correlated with its market price.

Log-Transformed Kilometers Driven (log\_kms): The original kms\_driven feature had a highly skewed distribution due to outliers. A logarithmic transformation normalized the distribution:

car['log\_kms'] = np.log1p(car['kms\_driven'])

This reduces the effect of extreme values, making the feature more suitable for linear models while preserving relative rankings.

Column Transformation: - Categorical features like name, company, and fuel\_type were converted into numerical representations using One-Hot Encoding (OHE). A column transformer was used to apply encoding only to categorical columns, leaving numerical features unchanged:

from sklearn.preprocessing import OneHotEncoder

from sklearn.compose import make\_column\_transformer

ohe = OneHotEncoder(handle\_unknown='ignore')

ohe.fit(X[['name','company','fuel\_type']])

column\_trans = make\_column\_transformer(

(OneHotEncoder(categories=ohe.categories\_), ['name','company','fuel\_type']),

remainder='passthrough' # numeric columns like car\_age and log\_kms remain unchanged

)

This ensures all categorical variables are machine-readable while retaining numerical features in their original form. Using a column transformer also streamlines the workflow, making it compatible with pipelines for any regression model.

Feature Selection: - Features were selected based on relevance and predictive potential:

Numerical Features: car\_age – captures depreciation effect, log\_kms – reduces skew in usage data.

Categorical Features: name – specific car model can significantly affect price, company – brand influence on pricing, fuel\_type – pricing trends vary by fuel type.

All features were retained, as they provide important information for predicting car prices. Tree-based models (Random Forest, XGBoost) can handle high-dimensional encoded features efficiently.

1. **Model Training, Evaluation and Selection: -**

Model Training: - Three regression models were trained on the processed car dataset to predict vehicle prices:

Linear Regression (LR): -

* + A baseline model to understand the linear relationship between features and price.
  + The dataset was split into training and testing sets using train\_test\_split.
  + One-Hot Encoding (OHE) was applied to categorical features (name, company, fuel\_type) using a column transformer.
  + A pipeline was created to combine encoding and regression into a single workflow:

pipe = make\_pipeline(column\_trans, LinearRegression())

pipe.fit(X\_train, y\_train)

* + Random states for the train-test split were tested iteratively, and the best random state was selected to maximize performance.

Random Forest Regressor (RFR)

* + An ensemble tree-based model capturing non-linear relationships and interactions between features.
  + Initial parameters: 200 trees, max depth 15.
  + Hyperparameter tuning was performed using GridSearchCV to optimize n\_estimators, max\_depth, min\_samples\_split, and min\_samples\_leaf.
  + A pipeline similar to LR was used to incorporate column transformation.

XGBoost Regressor (XGB)

* + A gradient boosting model designed for high predictive performance.
  + Parameters such as n\_estimators, learning\_rate, max\_depth, subsample, and colsample\_bytree were tuned using RandomizedSearchCV with repeated K-Fold cross-validation.
  + A pipeline was used to include column transformation for consistency.

Model Evaluation: - The models were evaluated using R² score on the test set:

| **Model** | **R² Score (Test)** |
| --- | --- |
| Linear Regression | 0.87 |
| Random Forest Regressor | 0.75 |
| XGBoost Regressor | 0.86 |

Observations:

* Linear Regression achieved the highest R² of 0.87, indicating a strong linear relationship between features and car price.
* XGBoost performed closely with 0.86, demonstrating its ability to capture non-linear patterns.
* Random Forest, even after hyperparameter tuning, performed slightly worse (0.75), likely due to overfitting on a small dataset.

Model Selection: - Selected Model: Linear Regression

* Reason: It achieved the highest test R² score (0.87) and is simpler and more interpretable than tree-based methods for this dataset.
* The model pipeline, including column transformation and feature engineering, was serialized using pickle for future use:

import pickle

pickle.dump(pipe, open('LinearRegressionModel.pkl', 'wb'))

Prediction on New Data: - New inputs can be predicted by applying the same feature engineering as the training data (car age and log-transformed kilometers) and feeding the processed data into the saved pipeline:

new\_data['car\_age'] = 2025 - new\_data['year']

new\_data['log\_kms'] = np.log1p(new\_data['kms\_driven'])

X\_new = new\_data[['name','company','car\_age','log\_kms','fuel\_type']]

prediction = pipe.predict(X\_new)

This ensures that all preprocessing steps are consistent between training and inference, resulting in reliable price predictions.

1. **Model Deployment: -** In this phase, the trained Linear Regression model is made available for real-time use. Users or applications can send input car details and receive immediate price predictions. The deployment is implemented using FastAPI, a modern Python web framework that allows us to expose the model as an API service.

**Why FastAPI?**

Expose the model as a service: Users or applications don’t need Python code to get predictions. They just send data to a REST endpoint.

Real-time predictions: The model can predict car prices immediately for new inputs.

Scalability & maintainability: FastAPI can handle multiple requests, integrate with web or mobile apps, and separate the model logic from API logic.

Input validation: FastAPI works with Pydantic to validate incoming data.

Step 1: Create FastAPI Application

Create a new Python file called app.py and import the necessary libraries:

from fastapi import FastAPI

from pydantic import BaseModel

import pandas as pd

import pickle

import numpy as np

FastAPI: Framework to create API endpoints.

Pydantic BaseModel: Defines the input data structure and performs validation.

pandas: Converts input into a DataFrame for the model.

numpy: Handles mathematical transformations (e.g., log transformation of kms\_driven).

pickle: Loads the trained Linear Regression pipeline.

Step 2: Load the Trained Model

Load the trained pipeline, which already includes preprocessing (One-Hot Encoding) and the regression model:

pipe = pickle.load(open(r"C:\Users\Kumarjit Brahma\LinearRegressionModel.pkl", "rb"))

Step 3: Initialize the FastAPI

app = FastAPI(title="Car Price Prediction API",

description="Predict car prices using a trained Linear Regression model with preprocessing pipeline.", version="1.0")

title: Sets the name of the API.

description: Provides a short summary of what the API does.

version: Specifies the version of the API.

Step 4: Define the Input Schema

Use Pydantic to define the expected input structure:

class CarFeatures(BaseModel):

name: str

company: str

year: int

kms\_driven: int

fuel\_type: str

This ensures users provide all required fields with the correct data types.

Step 5: Create the Prediction Endpoint

Define a POST endpoint /predict to handle incoming car data and return predictions:

@app.post("/predict")

def predict(car: CarFeatures):

"""

Predict the price of a car given its features.

Parameters:

car (CarFeatures): Input car details.

Returns:

dict: Predicted car price.

"""

# Convert input to DataFrame

df = pd.DataFrame([car.dict()])

# Apply feature engineering consistent with training

df['car\_age'] = 2025 - df['year'] # Replace 2025 with current year if needed

df['log\_kms'] = np.log1p(df['kms\_driven'])

# Select features used during training

X = df[['name', 'company', 'car\_age', 'log\_kms', 'fuel\_type']]

# Make prediction using the pipeline

prediction = pipe.predict(X)[0]

# Return prediction

return {"predicted\_price": round(prediction, 2)}

POST endpoint: Users send data in the request body.

pipe.predict(X)[0]: Predicts car price using the trained pipeline.

Step 6: Run the FastAPI App

Open a terminal in the folder containing app.py.

Install Uvicorn (ASGI server for FastAPI) if not already installed:

pip install "uvicorn[standard]"

Start the FastAPI server:

python -m uvicorn app:app --reload

--reload allows automatic server reload on code changes.

Step 7: Access Swagger UI

Open a browser and go to:

<http://127.0.0.1:8000/docs>

Swagger UI provides an interactive interface to test your API.

Example input for /predict:

{

"name": "Maruti Suzuki Swift",

"company": "Maruti",

"year": 2019,

"kms\_driven": 100,

"fuel\_type": "Petrol"

}

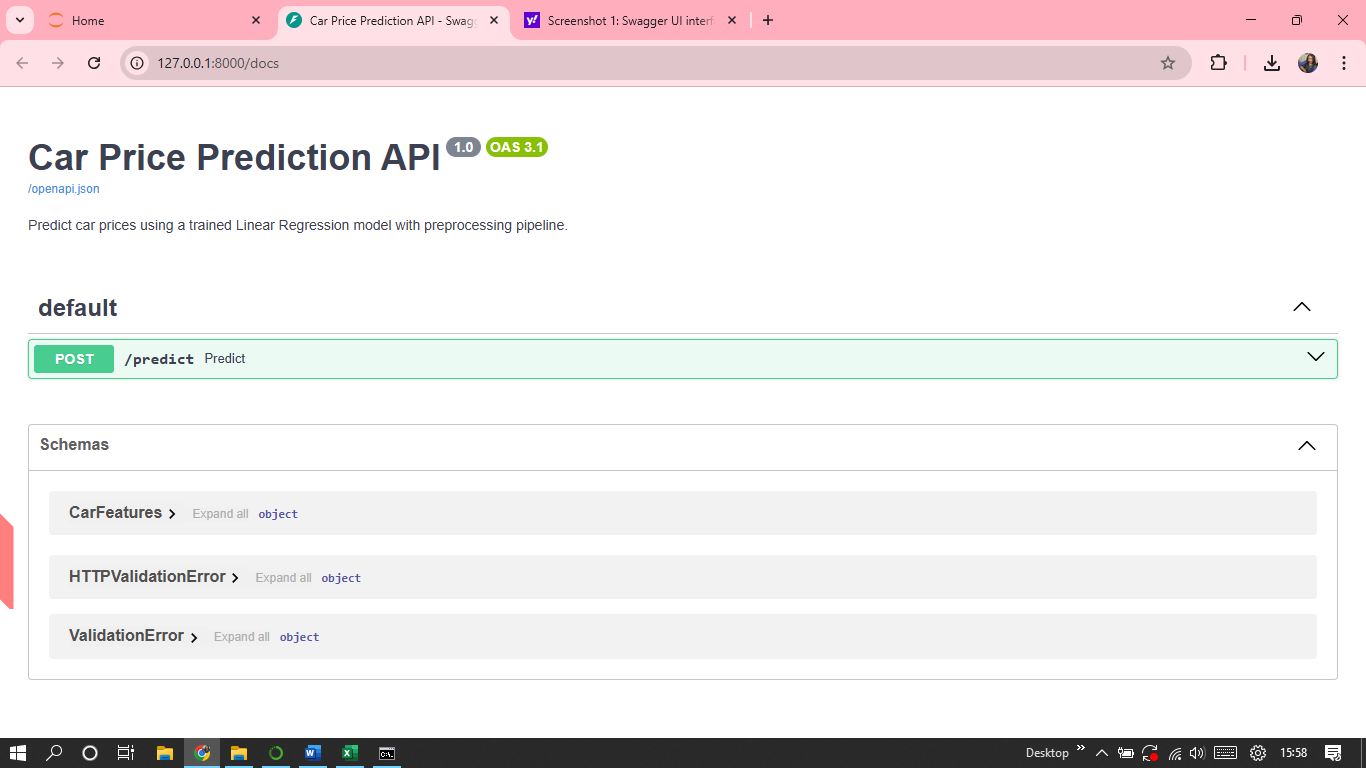
Click Execute → Output:

{

"prediction": 455275.79

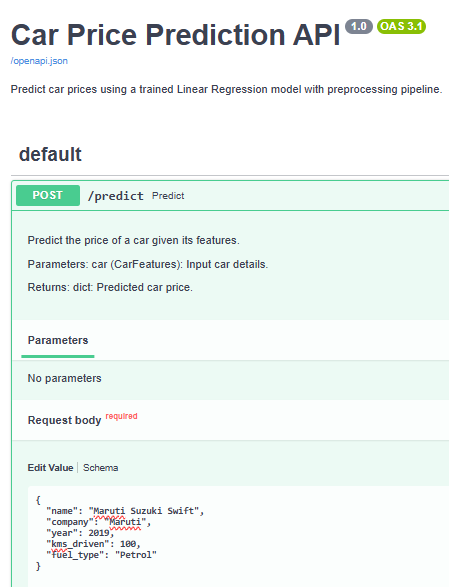
}

Screenshot 1: Swagger UI interface with /predict endpoint.

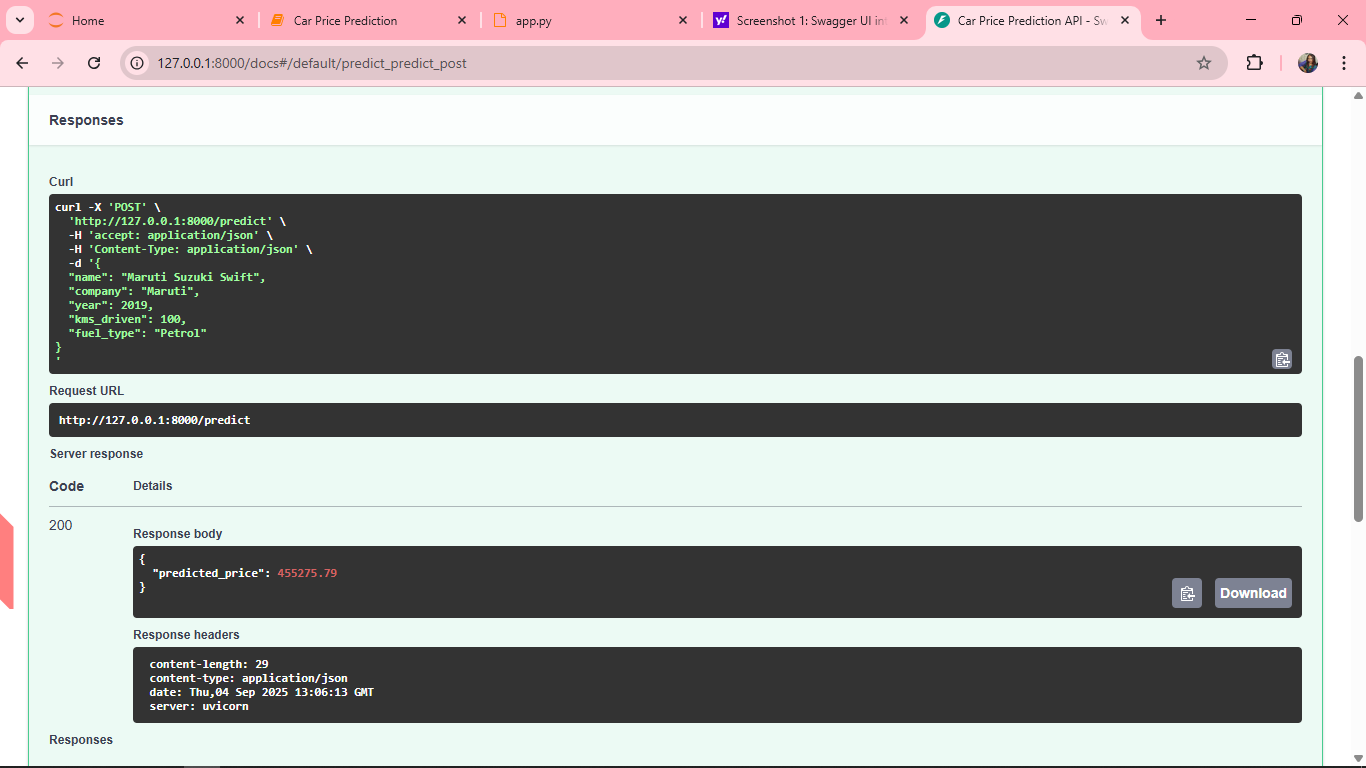


Screenshot 2: Example JSON input and output response.

Input



Output



Step 7: How It Works Internally

User sends JSON input to /predict.

FastAPI converts it into a CarFeatures object (validation).

Input is converted into a pandas DataFrame.

The pipeline (pipe) preprocesses data (One-Hot Encoding) and predicts the price.

FastAPI returns the prediction as JSON.

Conclusion

The deployed API allows real-time, automated car price predictions.

FastAPI ensures the system is scalable, maintainable, and easily integrated into web or mobile applications.

This separation of model training and deployment layer makes the solution production-ready.

This project successfully developed and deployed a Car Price Prediction system using the Quikr Car dataset, progressing through a complete ML lifecycle: from problem framing, data cleaning, and feature engineering to model training, deployment with FastAPI, beta testing, and optimization. Linear Regression emerged as the best-performing model, offering both accuracy and interpretability, while deployment ensured real-time predictions via an API endpoint. Key learnings include the importance of rigorous data preprocessing, the role of exploratory analysis in guiding feature engineering, and the value of automation and retraining in keeping models relevant over time. Beyond technical implementation, this project demonstrated the necessity of scalability, usability, and continuous improvement for ML solutions to deliver real business impact.