**VinAudit Project**

Document BY

Iman Imtiaz

2024

**Contents**

[Objective 4](#_Toc181924965)

[1. Design Document: Database Schema and Data Flow 4](#_Toc181924966)

[1.1 Dealers Table 4](#_Toc181924967)

[1.2 Vehicles Table 4](#_Toc181924968)

[1.3 Vehicle\_Attributes Table 5](#_Toc181924969)

[1.4 Listings Table 5](#_Toc181924970)

[1.5 Status\_Log Table 5](#_Toc181924971)

[1.6 ER Diagram 6](#_Toc181924972)

[2. Data Flow 6](#_Toc181924973)

[2.1 Input Data 6](#_Toc181924974)

[2.2 Database Query 7](#_Toc181924975)

[2.3 Pricing Calculation 7](#_Toc181924976)

[2.4 Output 8](#_Toc181924977)

[3. Setup Database 8](#_Toc181924978)

[3.1 Prerequisites 8](#_Toc181924979)

[3.2 Database and Tables Creation 9](#_Toc181924980)

[3.3 Data Cleansing 9](#_Toc181924981)

[3.4 Updating Listing Prices 10](#_Toc181924982)

[3.5 File Setup 10](#_Toc181924983)

[3.6 Script Workflow and Explanation 10](#_Toc181924984)

[3.7 Running the Script 11](#_Toc181924985)

[3.8 Expected Output 11](#_Toc181924986)

[3.9 Error Handling 11](#_Toc181924987)

[4. Backend Implementation 12](#_Toc181924988)

[4.1 Prerequisites 12](#_Toc181924989)

[4.2 Backend Code Creation 12](#_Toc181924990)

[4.3 Database Query for Matching Results 13](#_Toc181924991)

[4.4 Mileage-Based Price Prediction Logic 14](#_Toc181924992)

[4.5 Response Formatting 15](#_Toc181924993)

[4.6 API Request Example 17](#_Toc181924994)

[5. Frontend Implementation 17](#_Toc181924995)

[5.1 Prerequisites 17](#_Toc181924996)

[5.2 File Structure 17](#_Toc181924997)

[5.3 HTML and JavaScript Files Creation 17](#_Toc181924998)

[5.4 Running the Frontend 20](#_Toc181924999)

[6. Testing 20](#_Toc181925000)

[6.1 Test Case Documentation 20](#_Toc181925001)

[6.2 Error Handling 21](#_Toc181925002)

[7. Creating Virtual Environment and Deploying the Application 22](#_Toc181925003)

[7.1 Install Application Dependencies 22](#_Toc181925004)

[7.2 Create a WSGI Entry Point 22](#_Toc181925005)

[7.4 Configure Nginx 23](#_Toc181925006)

[7.5 Configure SELinux (If Enabled) 24](#_Toc181925007)

[7.6 Configure the Firewall 24](#_Toc181925008)

[7.7 Connect to VM via SSH and Transfer Files 25](#_Toc181925009)

[7.8 Set Up the Virtual Environment 25](#_Toc181925010)

[7.9 Test and Start Nginx: 26](#_Toc181925011)

[7.10 Running the Backend and Frontend 26](#_Toc181925012)

[8. Additional Improvements (Bonus) 28](#_Toc181925013)

[8.1 Additional Factors for Price Accuracy 28](#_Toc181925014)

[8.1.1 Vehicle Condition (Excellent, Fair, Poor) 28](#_Toc181925015)

[8.1.2 Location-Based Price Adjustments 29](#_Toc181925016)

[8.1.3 User-Defined Mileage Range 30](#_Toc181925017)

[8.1.4 Certified Pre-Owned (CPO) Status Impact on Pricing 30](#_Toc181925018)

[8.2 Enhanced UI: Additional Filters for Vehicle Attributes 30](#_Toc181925019)

[8.2.1 Filter by Trim Level 30](#_Toc181925020)

[8.2.2 Filter by Style (e.g., SUV, Sedan) 31](#_Toc181925021)

[8.2.3 Filter by Fuel Type (e.g., Gasoline, Electric, Hybrid) 31](#_Toc181925022)

[8.3 Sorting Results by Price or Mileage 31](#_Toc181925023)

**VinAudit Documentation**

# Objective

A smart search interface was developed to estimate the average market value for a vehicle based on its year, make, model, and optional mileage. This system included:

1. A web-based UI for input and results display.
2. A backend that calculates the average price of similar listings.
3. Database integration to store and retrieve data for cars and their listings.

# 1. Design Document: Database Schema and Data Flow

*The database schema design provided a structured foundation for storing, organizing, and retrieving application data. Below is a summary of the implemented tables in the MySQL database, their structures, and relationships.*

## 1.1 Dealers Table

**Purpose**: This table stored essential information about car dealers.

**Attributes**:

* + dealer\_id: Primary key, uniquely identifying each dealer.
  + dealer\_name: Dealer's name.
  + dealer\_street: Dealer’s street address.
  + dealer\_city: City of the dealer.
  + dealer\_state: State of the dealer.
  + dealer\_zip: ZIP code of the dealer’s location.

## 1.2 Vehicles Table

**Purpose**: This table contained comprehensive details about each vehicle available for sale.

**Attributes**:

* + vin: Primary key, Vehicle Identification Number for unique identification.
  + year: Year of vehicle manufacture.
  + make: Vehicle’s manufacturer (e.g., Toyota, Ford).
  + model: Specific model of the vehicle (e.g., Camry, F-150).
  + trim: Trim level of the vehicle (e.g., LE, XLT).
  + driven\_wheels: Type of drivetrain (e.g., FWD, RWD, AWD).
  + used: Boolean indicating if the vehicle is used or new.
  + certified: Boolean indicating if the vehicle is certified pre-owned.
  + dealer\_id: Foreign key linking to Dealers table, associating each vehicle with a dealer.

## 1.3 Vehicle\_Attributes Table

**Purpose**: This table included additional attributes to provide more detailed vehicle information.

**Attributes**:

* + vin: Foreign key linking to Vehicles table, aligning attributes with the corresponding vehicle.
  + engine: Engine type description (e.g., V6, Electric).
  + fuel\_type: Type of fuel (e.g., Gasoline, Diesel, Electric).
  + exterior\_color: Exterior color of the vehicle.
  + interior\_color: Interior color of the vehicle.
  + style: Vehicle’s body style (e.g., Sedan, SUV, Coupe).

## 1.4 Listings Table

**Purpose**: This table held data about vehicle listings, connecting vehicles to their sales availability.

**Attributes**:

* + listing\_id: Primary key, unique identifier for each listing.
  + vin: Foreign key linking to the Vehicles table.
  + dealer\_id: Foreign key linking to the Dealers table, indicating the dealer listing the vehicle.
  + listing\_price: Listed sale price.
  + listing\_mileage: Mileage at listing time.
  + seller\_website: Dealer’s website URL where the listing can be accessed.

## 1.5 Status\_Log Table

**Purpose**: This table tracked the status of each listing over time for historical analysis.

**Attributes**:

* + listing\_id: Foreign key linking to Listings table.
  + first\_seen\_date: Date when the listing was first observed.
  + last\_seen\_date: Date when the listing was last observed.
  + status: Current status of the listing (e.g., Active, Sold, Expired).

## 1.6 ER Diagram

|  |
| --- |
|  |

|  |
| --- |
|  |

# 2. Data Flow

*The data flow process demonstrated how data moved within the system, from initial user input to database interaction and output display.*

## 2.1 Input Data

**User Input**: The data flow began when users specified their vehicle search criteria, including:

* + year: Desired vehicle model year.
  + make: Manufacturer.
  + model: Specific model name.
  + mileage: Expected vehicle mileage.

|  |
| --- |
|  |

## 2.2 Database Query

**Search Process**: Based on user input, the application constructed and executed a database query to locate matching records in Vehicles and Listings tables. The query retrieved:

* + Vehicles matching the specified year, make, and model.
  + Relevant listings for the identified vehicles, including details such as pricing and mileage.

## 2.3 Pricing Calculation

**Price Estimation**: After fetching relevant listings, the application computed the average market price for the filtered listings, which included:

* + **Regression Analysis**: If mileage is specified, regression analysis was performed to adjust for factors like mileage, ensuring a more precise market value estimate.

## 2.4 Output

**Display Results**: The final output displayed:

* + **Estimated Market Value**: Average price derived from filtered listings and any regression analysis.
  + **Relevant Listings**: Up to 100 listings that met user criteria, presented in a user-friendly format showing make, model, price, mileage, and a link to the dealer’s website.

|  |
| --- |
|  |

# 3. Setup Database

*The MySQL database setup process was essential for the application's development. This setup involved creating the database, establishing the necessary tables, and populating them with data. The following steps detail the processes implemented.*

## 3.1 Prerequisites

**System Requirements**:

* + **MySQL Server**: Installed and running MySQL server.
  + **Python**: Installed (version 3.10 or later preferred).
  + **Required Libraries**: Installed necessary libraries for database interaction and data manipulation using pip.

***Required Libraries Installation*:**

Commands were executed to install the libraries:

**pip install mysql-connector-python pandas**

* **mysql-connector-python**: Installed mysql-connector==9.1.0 to connect Python to MySQL databases.
* **pandas**: Used for data manipulation and cleaning.

## 3.2 Database and Tables Creation

*File:* ***populate\_database.py***

This script connected to the MySQL server, created the VehicalData database, and established all required tables as per the schema. The script also loaded data into the database while handling any missing values and duplicates.

**Execution Steps**:

1. Update the script with MySQL credentials (username and password).
2. Save the code as populate\_database.py.
3. Execute the script in the terminal:

**python populate\_database.py**

*Outcome*: The script successfully created the database, structured tables, cleansed the data, handled missing values and duplicates, and populated the database with cleaned data.

## 3.3 Data Cleansing

*The data cleansing was implemented within the clean\_data function in populate\_database.py.*

* **Removing Missing Values**: Dropped rows with missing vin values, as VINs are essential for vehicle identification.
* **Removing Duplicates**: Retained only the latest occurrence of duplicate vin entries, ensuring unique entries in the database.

|  |
| --- |
|  |

## 3.4 Updating Listing Prices

*File:* ***update\_listing\_prices.py***

This script updated records in the Listings table where listing\_price values were missing or set to zero. The script was executed post data loading by populate\_database.py to ensure data consistency.

## 3.5 File Setup

*File*: Save the script as update\_listing\_prices.py in the same directory as other database scripts.

## 3.6 Script Workflow and Explanation

**Database Connection**: Established a connection to the MySQL database (VehicleData) using connect\_to\_mysql function, validating successful connection or printing an error message if unsuccessful.

**Identify Records with Zero Prices**: update\_zero\_listing\_prices fetched all records in the Listings table with listing\_price set to zero, estimating prices based on similar records.

**Query Logic for Price Estimation**:

* **Query 1**: Attempted to match listings by year, make, model, city, state, and similar mileage (±5%).
* **Query 2**: If no match, matched by year, make, model, and similar mileage.
* **Query 3**: If still unmatched, used year, make, and model only.
* Calculated average price from matches to update listing\_price where zero.

**Commit Changes**: Saved all modifications in the database.

**Close Database Connection**: Closed the connection post-operation.

## 3.7 Running the Script

**Execution Steps**:

1. Open a terminal.
2. Navigate to the directory containing update\_listing\_prices.py.
3. Execute the script:

**python update\_listing\_prices.py**

## 3.8 Expected Output

* **Output Messages**:
  + "Connected to MySQL database" confirmed successful connection.
  + "Listing prices updated where applicable" indicated completion of updates.

|  |
| --- |
|  |

## 3.9 Error Handling

* **Connection Issues**: Printed error message if database connection failed.
* **Schema Validation**: Errors were raised by MySQL if tables or columns were missing. Ensured database setup was accurate and contained all specified tables.

# 4. Backend Implementation

*The backend of the application was developed using Flask, a lightweight WSGI web application framework in Python. This section covers the setup of the Flask API server, the endpoints created, and the database queries implemented to retrieve vehicle listings and calculate estimated market values.*

## 4.1 Prerequisites

**System Requirements**:

* **Python**: Installed (version 3.10 or later preferred).
* **Required Libraries**: Installed Flask, scikit-learn 1.5.2, and mysql-connector-python for handling requests, performing linear regression, and connecting to the MySQL database.

***Required Libraries Installation*:**

Commands were executed to install the libraries:

**pip install Flask scikit-learn mysql-connector-python**

**Flask**: Used for setting up the web server and handling API requests.

**scikit-learn**: Provided tools for machine learning, including linear regression.

**mysql-connector-python**: Enabled Python to connect to MySQL databases.

## 4.2 Backend Code Creation

*File:* ***app.py***

This script established the Flask server and defined the API endpoint for handling incoming requests.

**Code Explanation**:

1. **Imports**: Imported necessary libraries, including Flask 3.0.3, MySQL connector 9.1.0, and scikit-learn 1.5.2 for linear regression.
2. **Database Connection Function**: Implemented get\_db\_connection() to establish a connection to the MySQL database.
3. **Flask API Endpoint**:
   * **Endpoint:** Defined the endpoint /, which accepted GET requests.
   * **Request Parameters:** Retrieved the year, make, model, and optional mileage parameters from incoming requests.
   * **Query Execution:** Queried the database to retrieve relevant listings based on provided parameters.
   * **Price Adjustment:** Used linear regression to adjust prices based on mileage, if provided.
   * **Response:** Returned a JSON response with the estimated market value and relevant listings.
4. **Error Handling**: Implemented error handling for missing parameters and database errors.

**Execution Steps**:

1. Save the code as app.py.
2. Replace MySQL credentials (username and password) in the script.
3. Execute the Flask application in the terminal:

**python app.py**

By default, the Flask server ran on <http://127.0.0.1:5000/>.

## 4.3 Database Query for Matching Results

**Function**: get\_listings(self, year, make, model, mileage=None)

This function fetched vehicle listings based on user-specified parameters, adjusting the query logic based on whether mileage was provided.

**Without Mileage Parameter**:

* Executed a base query to fetch listings for the specified year, make, and model where listing\_price is greater than zero.
* Returned up to 100 results.

**SQL Query**:

SELECT v.year, v.make, v.model, v.trim, l.listing\_price,   
 l.listing\_mileage, d.dealer\_city, d.dealer\_state  
FROM Vehicles v  
INNER JOIN Listings l ON v.vin = l.vin  
INNER JOIN Dealers d ON l.dealer\_id = d.dealer\_id  
WHERE v.year = %s  
 AND v.make = %s  
 AND v.model = %s  
 AND l.listing\_price > 0  
LIMIT 100;

**With Mileage Parameter**:

* Applied an additional ordering to prioritize listings based on proximity to the specified mileage using ABS(l.listing\_mileage - %s) ASC, which ordered results by absolute mileage difference.
* This approach ensured that listings with mileage closest to the specified input appeared first.

**SQL Query**:

SELECT v.year, v.make, v.model, v.trim, l.listing\_price,   
 l.listing\_mileage, d.dealer\_city, d.dealer\_state  
FROM Vehicles v  
INNER JOIN Listings l ON v.vin = l.vin  
INNER JOIN Dealers d ON l.dealer\_id = d.dealer\_id  
WHERE v.year = %s  
 AND v.make = %s  
 AND v.model = %s  
 AND l.listing\_price > 0  
ORDER BY ABS(l.listing\_mileage - %s) ASC  
LIMIT 100;

**Conversion of Results**: Converted Decimal types to float for listing\_price and listing\_mileage to ensure data compatibility in the JSON response.

## 4.4 Mileage-Based Price Prediction Logic

**Function**: predict\_price\_with\_mileage(listings, input\_mileage)

This function predicted the price for a vehicle based on mileage using linear regression. It allowed more accurate market value estimation by analyzing the relationship between mileage and price from existing listings.

**Logic Explanation**:

* Extracted mileage and price data from listings and converted them into arrays.
* Trained a LinearRegression model using mileage as the predictor and price as the response.
* Predicted the price for the specified input\_mileage.
* Rounded the predicted price to the nearest hundred for a cleaner presentation.

**Code**:

mileages = np.array([listing['listing\_mileage'] for listing in listings]).reshape(-1, 1)  
prices = np.array([listing['listing\_price'] for listing in listings])  
  
model = LinearRegression()  
model.fit(mileages, prices)  
  
predicted\_price = model.predict(np.array([[input\_mileage]]))  
return round(predicted\_price[0], -2)

## 4.5 Response Formatting

This logic formatted the JSON response for the API, displaying the estimated market value and relevant listings in a user-friendly manner.

* **With Mileage Parameter**:
  + Calculated an estimated price using predict\_price\_with\_mileage() for the provided mileage.
  + Returned the estimated average price along with a list of matching listings.
* **Without Mileage Parameter**:
  + Returned only the list of listings and an average price for the results.

**Sample Response Structure**:

|  |
| --- |
|  |

## 4.6 API Request Example

*To test the API endpoint, the following GET request was used in a web browser*:

<http://127.0.0.1:5000/?year=2015&make=Kia&model=FORTE&mileage=53960>

*Outcome*: The request returned a JSON response with the estimated market value and a list of relevant listings.

# 5. Frontend Implementation

*The frontend of the application was developed using HTML and JavaScript to create an interface allowing users to input search parameters and view results. The following sections outline the structure of these files, the steps to run them, and the complete interaction flow between the frontend and backend.*

## 5.1 Prerequisites

* **System Requirements**:
  + A modern web browser, such as Chrome, Firefox, or Edge, to view and test the HTML files.
  + A local or remote server to serve the HTML files.

## 5.2 File Structure

The directory structure was organized as follows for the frontend files:

├── index.html  
├── results.html  
└── script.js

## 5.3 HTML and JavaScript Files Creation

#### **File: index.html**

*This file served as the search page, allowing users to input their search criteria.*

* **Code Explanation**:
  + The HTML structure included a form with fields for user inputs (year, make, model, and mileage).
  + A button was provided to submit the form, triggering a search.
  + A <div> with the ID results was included to display search results after an AJAX call.
* **URL:**

<http://127.0.0.1:5000/static/index.html>

|  |
| --- |
|  |

#### **File: results.html**

*This file displayed the results returned from the backend.*

* **Code Explanation**:
  + The results.html file contained a section to display the estimated market value and a table for relevant listings.
  + The market-value and listings-table-body IDs were used to dynamically populate data received from the backend.

|  |
| --- |
|  |

#### **File: script.js**

*This JavaScript file managed the form submission, sent an AJAX request to the backend, and processed the response to display results.*

* **Code Explanation**:
  + An event listener was added to handle the form submission.
  + Default submission was prevented, and input values were retrieved.
  + An AJAX request was sent using the Fetch API to the backend server with a query string containing user inputs.
  + Upon receiving a successful response, the estimated market value and relevant listings were extracted and displayed in the corresponding HTML elements.
  + Error handling was included to manage any issues during the AJAX request.

## 5.4 Running the Frontend

**Execution Steps**:

1. Save the HTML and JavaScript files:
   1. Create index.html, results.html, and script.js files in the main directory with the corresponding code.
2. **Running a Local Server**:
   1. Used Python’s built-in HTTP server to serve the static HTML files by navigating to the frontend directory in the terminal and running:

**python -m http.server**

* 1. This command started a local server on port 8000.

1. **Accessing the Application**:
   1. Opened a web browser and navigate to <http://127.0.0.1:8000/index.html> to access the search page.
   2. Filled in the fields and submitted the form to view the results populated on the same page.

# 6. Testing

*Testing was a critical phase of the application development, ensuring that all components functioned as intended and provided a smooth, error-free user experience. This section outlines the process for creating test cases, executing manual tests, and managing potential errors.*

## 6.1 Test Case Documentation

*To effectively organize and track test scenarios, a Google Sheet was created to outline various test cases, including valid and invalid inputs, expected outcomes, and actual outcomes.*

**Google Sheet Setup**

1. Created a Google Sheet named "Vehicle Market Value Estimator Test Cases." At [Vehicle Market Value Estimator Test Cases - Google Sheets](https://docs.google.com/spreadsheets/d/1OFB7ugODLnyD0mCo3kBTJTtTpVmN-dC70reIB1OFW0U/edit?gid=0#gid=0)
2. Set up columns to capture essential information:
   * **Test Case ID**: Unique identifier for each test case.
   * **Input Parameters(Year, Make, Model, Mileage)**: Values entered in the search form.
   * **Expected Price with Linear Regression**: Predicted outcome based on the linear regression of mileage.
   * **Expected Price with Linear Regression**: Predicted outcome based on the average of 100 closet records.

## 6.2 Error Handling

*Effective error handling was implemented to create a robust application by managing missing input values, handling empty results, and addressing database connectivity issues.*

**Error Handling Strategies**

1. **Missing Input Values**:
   * **Implementation**: In script.js, checks were added to validate that all required input fields were filled before sending an AJAX request.
   * **User Feedback**: If required fields were missing, an error message appeared above the form indicating which fields needed input.
2. **Empty Results**:
   * **Implementation**: On the backend (Flask server), cases with no listings matching the query were handled.
   * **User Feedback**: A message indicates that no results matched the criteria:
     + "No listings found for the specified criteria."
3. **Database Connectivity Issues**:
   * **Implementation**: try-except blocks were implemented in the backend to catch exceptions related to database connectivity.
   * **User Feedback**: If a connection issue occurred, a user-friendly error message was returned:
     + "Database connection error. Please try again later."

# 7. Creating Virtual Environment and Deploying the Application

*Creating a virtual environment and deploying the application were essential steps to isolate dependencies, set up the application server, and ensure it ran reliably. The following sections outline each step in this process.*

## 7.1 Install Application Dependencies

To set up the required dependencies, commands were executed within the virtual environment.

1. **Install Flask and Gunicorn**:

**pip install flask gunicorn**

1. **Install from Requirements File:**

**pip install -r requirements.txt**

## 7.2 Create a WSGI Entry Point

A WSGI entry point was created to connect the Flask application to the WSGI server (Gunicorn).

1. **Create wsgi.py**:

**nano wsgi.py**

**7.3 Set Up a systemd Service**

A systemd service was configured to manage the application and ensure it ran as a background service.

1. **Create a systemd Service File**:

sudo nano /etc/systemd/system/flaskapp.service

1. **Add the Following Content**:

[Unit]

Description=Gunicorn instance to serve Flask application

After=network.target

[Service]

User=root

WorkingDirectory=/var/www/flaskapp

Environment="PATH=/var/www/flaskapp/venv/bin"

ExecStart=/var/www/flaskapp/venv/bin/gunicorn --workers 3 --bind 0.0.0.0:8000 wsgi:app

[Install]

WantedBy=multi-user.target

1. **Start and Enable the Service**:

sudo systemctl start flaskapp

sudo systemctl enable flaskapp

sudo systemctl status flaskapp

## 7.4 Configure Nginx

Nginx was set up as a reverse proxy to route requests to the Flask application.

1. **Create an Nginx Configuration File**:

**sudo nano /etc/nginx/conf.d/flaskapp.conf**

1. **Add the Following Configuration**:

nginx

server {

listen 80;

server\_name imantest.linkgrid.com;

location / {

proxy\_pass http://localhost:5000;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

}

}

1. **Test and Start Nginx**:

**sudo nginx -t**

**sudo systemctl start nginx**

**sudo systemctl enable nginx**

## 7.5 Configure SELinux (If Enabled)

To allow Nginx to connect to the application server, SELinux configuration was adjusted.

**sudo setsebool -P httpd\_can\_network\_connect 1**

## 7.6 Configure the Firewall

Firewall settings were adjusted to allow HTTP and HTTPS traffic.

1. **Open Ports for HTTP and HTTPS**:

**sudo firewall-cmd --permanent --add-service=http**

**sudo firewall-cmd --permanent --add-service=https # if HTTPS is needed**

1. **Reload the Firewall**:

**sudo firewall-cmd --reload**

*This process successfully created a virtual environment, installed dependencies, configured the application server, and set up Nginx as a reverse proxy for deployment.*

## 7.7 Connect to VM via SSH and Transfer Files

1. **SSH into the Virtual Machine**: Use SSH to connect to the virtual machine (VM) from the local machine:

**ssh root@192.206.41.180**

1. **Create a Directory for the Project on the VM**: After logging in, create a directory to store the project files:

**mkdir -p /var/www/webapp**

1. **Transfer Files from Local Machine to VM**: Use scp to copy files from the local machine to the VM. Replace username, vm\_ip\_address, and file paths with the actual values.

**scp -r /pathtoyourprojectfiles root@192.206.41.180:/var/www/ webapp**

This command will transfer all project files, including HTML, JavaScript, and Python scripts, to the specified directory on the VM.

## 7.8 Set Up the Virtual Environment

1. **Create a Virtual Environment**: Within the /var/www/webapp directory on the VM, create a virtual environment:

**cd var/www/webapp**

**python3 -m venv venv**

1. **Activate the Virtual Environment**:

**source venv/bin/activate**

1. **Install Application Dependencies**: Once inside the virtual environment, install the necessary dependencies:

**pip install flask gunicorn**

**pip install -r requirements.txt**

## 7.9 Test and Start Nginx:

**sudo nginx -t**

**sudo systemctl start nginx**

**sudo systemctl enable nginx**

## 7.10 Running the Backend and Frontend

1. **Run the Flask Backend**: The backend service, managed by gunicorn and systemd, was started with the commands in the systemd service setup. To verify, check the status:

**Pyhton populate\_database.py**

**Python update\_null\_prices.py**

**Pyhton app.py**

**sudo systemctl status flaskapp**

1. **Access the Backend**: The backend files can be accessed through at:

*https://imantest.linkgrid.com/?year=2015&make=Kia&model=FORTE&mileage=53960*

|  |
| --- |
|  |

1. **Access the Frontend**: The frontend files (e.g., index.html, results.html) can be accessed through the configured Nginx server at:

*https://imantest.linkgrid.com/static/index.html*

|  |
| --- |
|  |

# 8. Additional Improvements (Bonus)

*The following improvements can be implemented to enhance price accuracy, user control, and search specificity, resulting in a more robust and user-friendly application.*

## 8.1 Additional Factors for Price Accuracy

Incorporating additional factors into the pricing algorithm can improve the accuracy of estimated market values. These include vehicle condition, location-based adjustments, user-defined mileage ranges, and Certified Pre-Owned (CPO) status.

### 8.1.1 Vehicle Condition (Excellent, Fair, Poor)

* **Purpose**: Accounting for the vehicle's condition provide a nuanced price estimate, as the market value varied based on condition.
* **Implementation Steps**:
  1. **Database Schema Update**: Add a condition column to the Vehicles table with values like 'Excellent,' 'Fair,' and 'Poor'.

**ALTER TABLE Vehicles ADD COLUMN condition ENUM('Excellent', 'Fair', 'Poor') DEFAULT 'Fair';**

* 1. **Data Entry**: Ensure each vehicle record included a condition value.
  2. **Backend Modification**: Update the price estimation query to adjust for vehicle condition:

**condition\_adjustment = {**

**'Excellent': 1.1,**

**'Fair': 1.0,**

**'Poor': 0.9**

**}**

**estimated\_price \*= condition\_adjustment[vehicle\_condition]**

* 1. **Frontend UI**: Add a dropdown for users to select vehicle condition, passing the value to the backend.

### 8.1.2 Location-Based Price Adjustments

* **Purpose**: Adjust prices based on location to provide a more accurate local market estimate.
* **Implementation Steps**:
  1. **Location-Based Adjustments**: Define adjustment factors by location category.
  2. **Backend Modification**: Apply location-based factors in the pricing algorithm:

**location\_adjustment = {**

**'Urban': 1.05,**

**'Suburban': 1.0,**

**'Rural': 0.95**

**}**

**estimated\_price \*= location\_adjustment[location\_category]**

* 1. **User Input for Location**: Enable users to specify a location filter, refining searches by city or state.

### 8.1.3 User-Defined Mileage Range

* **Purpose**: Allow users to specify a mileage range for more specific searches.
* **Implementation Steps**:
  1. **Frontend UI**: Add fields for "Minimum Mileage" and "Maximum Mileage" on the search page.
  2. **Backend Modification**: Adjust the SQL query to filter listings within the user-defined mileage range:

**SELECT \* FROM Listings**

**WHERE year = %s AND make = %s AND model = %s**

**AND listing\_mileage BETWEEN %s AND %s;**

### 8.1.4 Certified Pre-Owned (CPO) Status Impact on Pricing

* **Purpose**: Apply a price increase for Certified Pre-Owned vehicles, given their higher value.
* **Implementation Steps**:
  1. **Database Schema Update**: Verify the presence of a certified column in the Vehicles table.
  2. **Backend Modification**: Adjust the price calculation for CPO vehicles:

**if is\_certified:**

**estimated\_price \*= 1.05**

* 1. **UI Modification**: Add a "Certified Pre-Owned" checkbox on the search page for users to filter only CPO vehicles.

## 8.2 Enhanced UI: Additional Filters for Vehicle Attributes

Enhancing the UI with more filters will allow users to perform highly specific searches, improving the user experience and relevance of search results.

### 8.2.1 Filter by Trim Level

* **Purpose**: Allow users to specify a trim level (e.g., LE, SE, XLE) for more precise results.
* **Implementation Steps**:
  1. **Frontend UI**: Add a "Trim Level" dropdown on the search form.
  2. **Backend Modification**: Filter listings based on the selected trim level:

**SELECT \* FROM Listings**

**WHERE year = %s AND make = %s AND model = %s AND trim = %s;**

### 8.2.2 Filter by Style (e.g., SUV, Sedan)

* **Purpose**: Allow users to filter by vehicle style, catering to preferences for body types.
* **Implementation Steps**:
  1. **Frontend UI**: Include a "Style" dropdown with options like "SUV," "Sedan," "Truck," etc.
  2. **Backend Modification**: Filter listings based on the selected style:

**SELECT \* FROM Listings**

**WHERE year = %s AND make = %s AND model = %s AND style = %s;**

### 8.2.3 Filter by Fuel Type (e.g., Gasoline, Electric, Hybrid)

* **Purpose**: Include fuel type as a filter to refine searches based on fuel preferences.
* **Implementation Steps**:
  1. **Frontend UI**: Add a "Fuel Type" dropdown with options like "Gasoline," "Diesel," "Electric," and "Hybrid."
  2. **Backend Modification**: Filter listings based on fuel type:

**SELECT \* FROM Listings**

**WHERE year = %s AND make = %s AND model = %s AND fuel\_type = %s;**

## 8.3 Sorting Results by Price or Mileage

**Purpose**: Allow users to sort the results by price or mileage in ascending (low-to-high) order, adding flexibility to the search experience.

**Implementation Steps**:

**Frontend UI**: Add a sorting dropdown for users to choose between "Sort by Price" and "Sort by Mileage" in ascending order.

**Backend Modification**: Modify the SQL query based on the user’s selected sort option:

* **Sort by Price**:

SELECT v.year, v.make, v.model, v.trim, l.listing\_price, l.listing\_mileage, d.dealer\_city, d.dealer\_state  
FROM Vehicles v  
INNER JOIN Listings l ON v.vin = l.vin  
INNER JOIN Dealers d ON l.dealer\_id = d.dealer\_id  
WHERE v.year = %s AND v.make = %s AND v.model = %s AND l.listing\_price > 0  
ORDER BY l.listing\_price ASC  
LIMIT 100;

* **Sort by Mileage**:

SELECT v.year, v.make, v.model, v.trim, l.listing\_price, l.listing\_mileage, d.dealer\_city, d.dealer\_state  
FROM Vehicles v  
INNER JOIN Listings l ON v.vin = l.vin  
INNER JOIN Dealers d ON l.dealer\_id = d.dealer\_id  
WHERE v.year = %s AND v.make = %s AND v.model = %s AND l.listing\_price > 0  
ORDER BY l.listing\_mileage ASC  
LIMIT 100;

This sorting feature allow users to prioritize listings based on either price or mileage, enhancing the overall user experience and usability of the application.