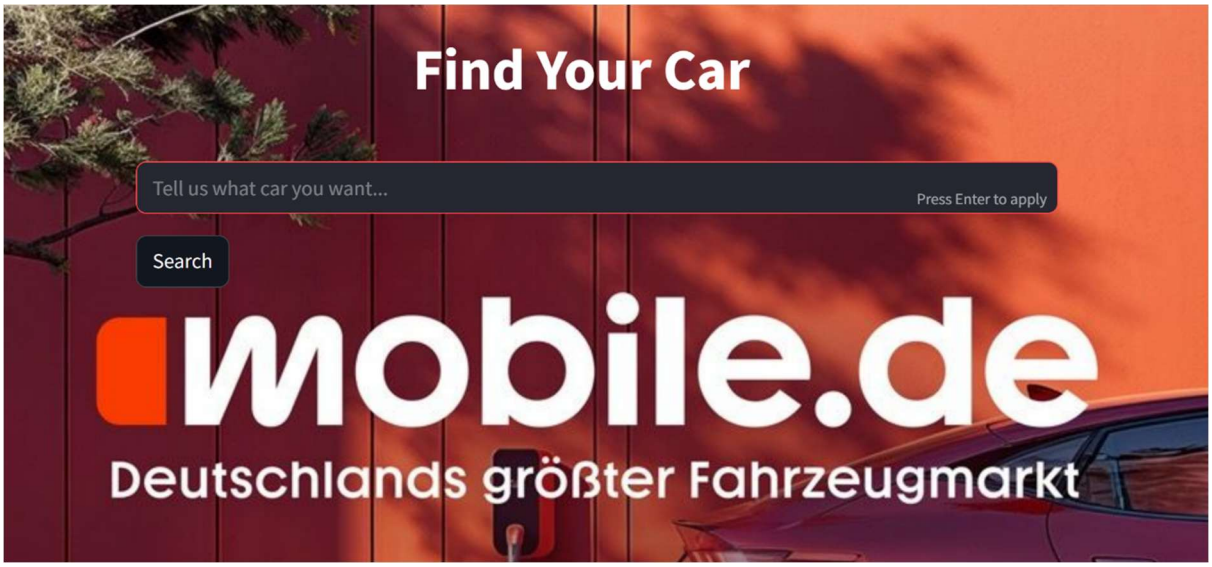
Find Your Car in Kiel:
A RAG-Based Recommendation System



1 Introduction

1.1 Motivation:

The pre-owned automobile market is vast and constantly changing. It provides customers with an effective alternative to purchasing new vehicles. Purchasing a vehicle is a high-stakes financial decision, involving mutual compromises between mileage, price, vehicle history and other technical specifications. A Buyer generally spends a lot of time getting the perfect car that they want. To make it easy for buyer, Mobile.de recently integrated an AI-powered search into their website. This feature allows users to find cars using human spoken language. In the present AI era, this strategic move will keep them a step ahead of their competitors, who still rely on traditional filter-based systems.

My motivation for this capstone Project comes directly from this industry advancement. Inspired by Mobile.de's modernised car search, this project aims to replicate this cutting-edge functionality on a smaller scale. In this capstone, I built a micro version of an AI car assistant using data from Kiel and explored the technical architecture required to implement the next generation of marketplace search.

1.2 Problem Statement

Current vehicle marketplaces rely on outdated filter systems that fail to understand human intent. And makes users spend a lot of time finding the right one for them from thousands of listings. This frustrates the buyers who know what they need but hard to find them in those thousands of listings.

1.3 Project Goal

The main goal of this capstone project is to build an RAG (Retrieval-Augmented Generation) System-based car recommendation assistant.

Specifically, the project aims to:

- **Use Local Data**: Scrape and process real-time listings from mobile.de around Kiel, Schleswig-Holstein region and test the system on real world data.
- **Replicate AI search logic**: Develop a local RAG pipeline that allow users to query in human language and get the expected car recommendations from the dataset.
- **Deploy locally with Streamlit**: Build a simple interactive web interface using Streamlit for the RAG engine that runs locally.

2 Data Acquisition (Web Scraping)

2.1 Data Source

The data for this project is scraped from mobile.de, Germany's largest vehicle marketplace. This website has a large number of used car listings and displays all the important car specifications, seller info and price. For the scope of this project, the data collection was focused on the region of Kiel, Schleswig-Holstein.

2.2 Technology Stack

To get the data, a custom web scraping pipeline was built using Python. Mobile.de provides the dynamic JavaScript content. I used "*Selenium*" library for rendering the website. The website often block standard Selenium WebDrivers. So, the "*undetected_chromedirver*" was used to initialise the browser instance and bypass the website's anti-bot detection systems.

2.3 The Scraping Pipeline

The data acquisition process followed a structured four-step workflow:

i. **Handling cookies pop-up**: Upon launching the browser using drivers, "*WebDriverWait*" is used to add a delay until it detects and clicks the cookie consent button, which is in class_name = *"mde-consent-accept-btn",* and also ensures that the button is clicked without overlaying issues.

ii. **Setting filters**: The code interacts with the location input field with id = *"geolocation-autosuggest"*, sets it to *"Kiel, Schleswig-Holstein"*, and return.

iii. **Data Extraction**: The function *"scrape_cars"* iterates through all the listings in the page using XPATH selectors to extract attributes in every listing container. The extracted field includes:
   - **Vehicle Model and Description**: extracted from the header tag *<h2>*.
   - **Price**: extracted from [*data-testid ="price-label"*] element.
   - **Specifications**: A list of attributes including first registration, mileage, power (kW/hp) and fuel type. Extracted from [*data-testid="listing-details-attributes"*]
   - **Seller information**: This list includes the seller's name, rating, and review count. Extracted from [*data-testid="seller-info"*]
   - **URL link**: extracted from *a[href*="details.html"]*

iv. **Pagination**: After scraping all the listings on the page, *"scrape_cars"* locates the "Next" button from [*data-testid="pagination:next"*]. If enabled, it clicks the button and waits for the next page to load. This process repeats until no further page is found.

3 Data Processing and Feature Engineering

3.1 Data cleaning

The raw scraped data from mobile.de has unstructured text fields, and some errors in the models name. These should be preprocessed to achieve a good RAG result.

As a part of data preprocessing, the following cleaning steps were implemented:

- **Model Name error**: On the initial inspection of the data, it was found that there are 153 listings with the model name "NEW", and the real model name is in the description. So, a logic was applied to swap the values of description to model while marking the description as "Not provided by seller".
- **Defensive Parsing**: Used "*safe_literal_eval*" function to handle the inherent variability of web-scrapped data. This helps the system from errors by processing list strings that appeared as equipment lists, without disrupting the pipeline during incorrect or missing data.
- **Parsing Price**: Parse price by removing all "," and convert into float datatype. (eg: 18,999(var) → 18999(float)).
- **Parsing Specifications**: Developed a function, "*Parse_specifications*", that extracts specific entities from long text. Used Regex to identify the format of "*first registration"* (MM/YYYY), isolate numeric values of "*mileage*" and "*power*", detect keywords for "*fuel type*", and mark "TRUE" when it comes across "*accident-free*".

- **Parsing Seller column**: As same as above, we developed a function, "*parse_seller*", that extracts seller details ("*seller_name*", "*seller_rating*", "*seller_reviews_count*", "*seller_pincode*" and "*seller_city*") using regex.
- **Normalisation Model and extract brand**: Function "*normalize_model*", delete any extra spaces and concat the car brand into a new column "*brand_model*".
- **Drop Duplicates**: Just drop any duplicates that are found, so that results will not repeat.

3.2 Feature Engineering

Apart from data cleaning, specific features that can be drawn are helpful to enhance the performance of the RAG application. Some derived features are:

- **Car age**: Function "*calc_age*" derives the car's age in years from the present date and "*first_registration*", which might be helpful for queries like "looking for a car not older than 5 years".

```
today = datetime.today()
years = today.year - dt.year - ((today.month, today.day) < (dt.month, dt.day))
```

- **Mileage Category**: Function "*mileage_cat*" groups the cars as <50k, 50k-100k, >100k categories. This helps for quick bucketing for the queries like "looking for a car with low mileage"

```
try:
    m = int(m)
    if m < 50000:
        return "<50k"
    if m <= 100000:
        return "50k-100k"
    return ">100k"
except Exception:
    return None
```

- **RAG Context Creation**: Function "*rag_text*" converts everything into a readable text. This was a crucial step for the AI component, making it easy to embed a single text. Embedding models expect a text chunk, not raw numeric fields. It also reduces hallucination and reads the actual numeric values like price and mileage.

  Cleaner RAG Context = Better Embeddings = Better Answer

  Sample rag_text:
  Mercedes-Benz | Mercedes-Benz CLA 200 Shooting Brake | First registration: 2017-09 | Accident-free: No | Mileage: 92,215 km | Fuel: Petrol | Power: 115.0 kW | Price: €18,999 | Seller: Kamux Auto GmbH (4.3 ⭐) | Description: 7G-DCT AMG Line+LED+NAVI

– **Data Serialisation**: Finally, after all the data cleaning and data engineering steps, the dataset was serialised into JSONL (JSON Lines) format. Unlike standard CSV or JSON arrays, JSONL allows for line-by-line streaming, which is a scalable standard for large language models.

Sample JSON Line:

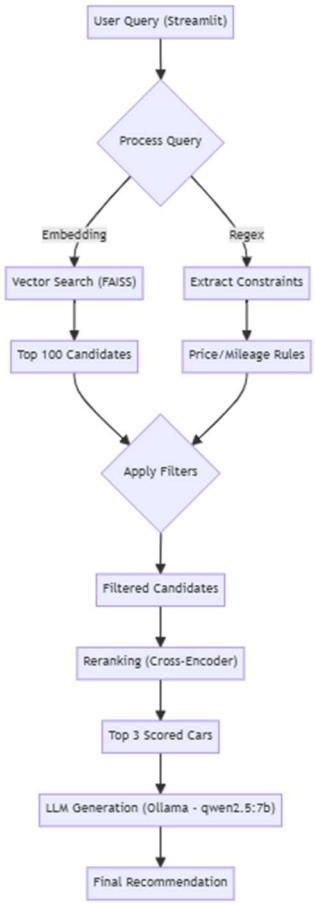
{"id": "car_2", "text": "Volkswagen | Volkswagen Tiguan | First registration: 2017-02 | Accident-free: No | Mileage: 88,855 km | Fuel: Petrol | Power: 132.0 kW | Price: €20,989 | Seller: Kamux Auto GmbH (4.6⭐) | Description: 2.0TSI 4Motion DSG Highline+LED+NAVI+HUD", "meta": {"brand": "Volkswagen", "model": "Volkswagen Tiguan", "price_eur": 20989, "first_registration": "2017-02", "mileage_km": 88855, "car_link": "https://suchen.mobile.de/fahrzeuge/details.html?id=441664842&cn=DE&dam=false&gn=Kiel%2C+Schleswig-Holstein&isSearchRequest=true&ll=54.322684%2C10.13586&od=up&rd=100&ref=srp&refId=692b2536-d65d-caf0-12e7-d05cb10a818e&s=Car&sb=rel&searchId=692b2536-d65d-caf0-12e7-d05cb10a818e&vc=Car", "accident_free": false, "car_age_years": 8, "mileage_category": "50k-100k", "power_kw": 132.0, "fuel_type": "Petrol", "seller_name": "Kamux Auto GmbH", "seller_rating": 4.6, "seller_reviews_count": 611}}

Each line contains a unique ID, text (rag_text), and a structured meta (metadata). *text* is for embedding, and the *meta* dictionary is for filtering.

4 System Architecture (The RAG engine)

4.1 Architectural overview

Retrieval Augmented Generation (RAG) is a design pattern that augments the capabilities of a chat completion model like ChatGPT by adding an information retrieval step, incorporating your proprietary enterprise content for answer formulation. For an enterprise solution, it's possible to fully constrain generative AI to your enterprise content.



The data flow operates in three distinct stages:

– **Ingestion & Indexing**: Raw scraped data is processed into vector embeddings and stored in a FAISS index. This helps in quick retrieval.

– **Hybrid Retrieval**: User's human language query is processed by a Semantic searcher and Rule-Based Parser at same time. Semantic searcher finds the relevant cars and rule- based parser extract strict price/mileage constraints.

– **Refinement & Generation**: The retrieved cars are further filtered using Cross-Encoder that re-scores for precision. Finally, passed to a Local LLM (Ollama - qwen2.5:7b) to generate a 3 cars recommendation with 2-3 lines of description on that recommendation.

4.2 Vector Retrieval (The semantic Layer)

The first stage coverts the user's query into a mathematical representation to identify the exact cars from the dataset.

- **Embedding Model**: "*all-MiniLM-L6-v2*" model from *SentenceTransformers* library is used for embedding. This model efficiently maps the dense "*rag_text*" of each car into a 384-dimensional vector space.
- **Indexing**: These vectors are indexed using FAISS (Facebook AI similarity Search) to get quick retrieval. The top 100 cars are retrieved semantically with "*InderFlatIP*" (Inner Product) index for every query.

4.3 Hybrid Filtering (The Constraint Layer)

There is a limitation for a pure vector, they are not able to strictly adhere to numerical boundaries (e.g., under 20k, around 20k). To solve this, a specific filtering layer interacts with the vector results:

- **Constraint Extraction**: Built an logic, "*extract_price_constraints*", with Regex that analyses the user's query to extract hard constraints. It detects the patterns for "*price*" (e.g., "under 30,000", "between 15k and 20k").
- **Metadata filtering**: These extracted rules are applied to the metadata of the top 100 vector cars. For example, if a user specifies a maximum price limit, any cars exceeding this limit are removed from the recommendation list.

4.4 Neural Reranking (The Precision Layer)

To refine the remaining cars further, a Cross-Encoder model ("*ms-marco-MiniLM-L-6-v2*") is used. It processes the user query and the dataset simultaneously. It assigns a precise relevance score to each filtered car and resort the list to ensure the best car matches for the next step (LLM).

4.5 Generation (The Reasoning Layer)

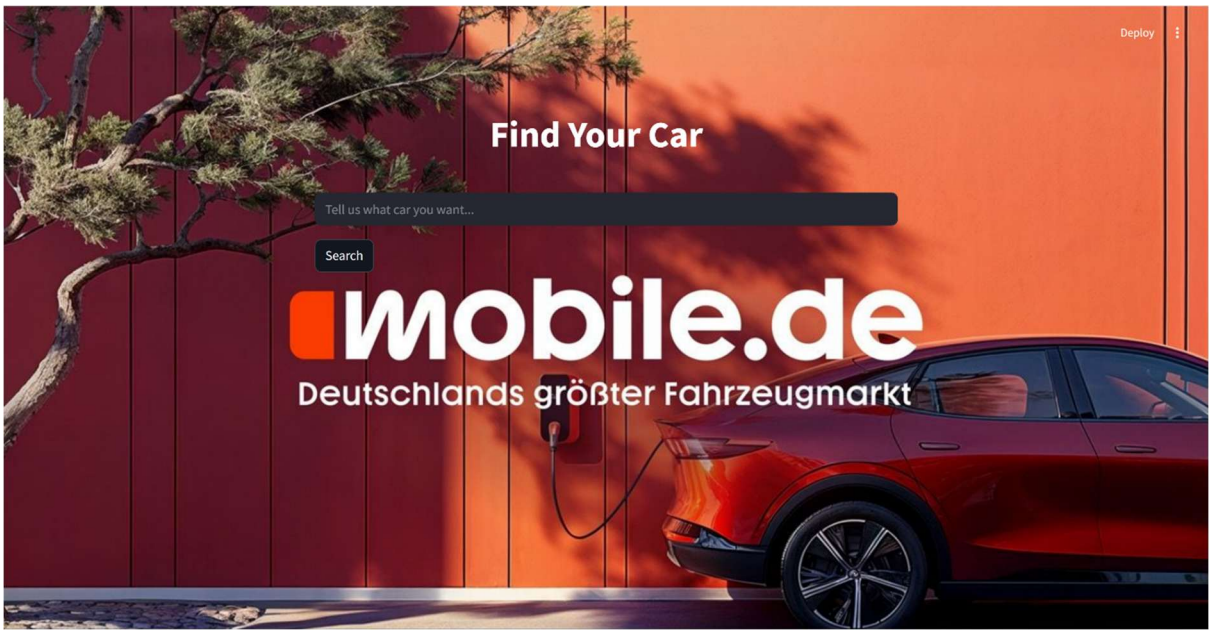
This is the final stage, where the top 3 car recommendations are synthesised into a natural language response.

- **Local LLM**: Ollama running the *qwen2.5:7b* model is integrated locally. This allows for online inference with API costs.
- **Prompt Engineering**: A structed prompt is feeded, that provides details of the top 3 cars of the LLM. This model is instructed to act as a "car recommendation assistant" and instructed to give 2-3 brief sentence explanation of why these specific cars fits to the user by minimizing the risk of hallucination.

5 User Interface

5.1 Design

To demonstrate the capabilities of the RAG backend, a simple user interface is developed using Streamlit. The main Purpose of this is to serve as a functional wrapper around the retrieval engine that allow users to interact.



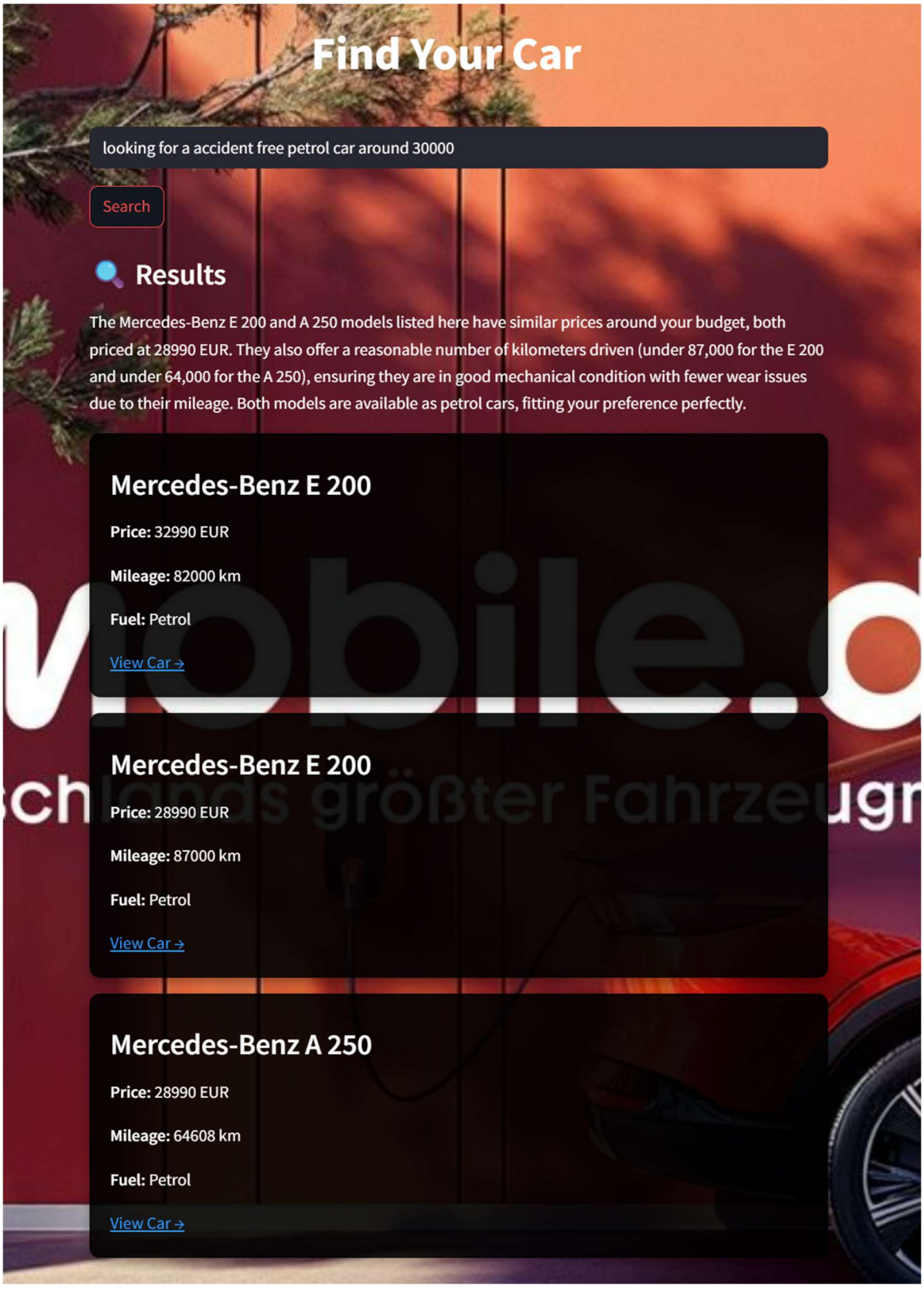
5.2 Implementation

This application acts as a direct frontend for the RAG engine.

- **Input**: A simple text box is provided where the user can enter their natural language query.
- **Processing**: After clicking on search button, this interface calls the backend with ask(query) function that passes the text through the entire RAG pipeline.
- **Output**:
  i. AI Reasoning: a text block displays the LLM's explanation for the selection.
  ii. Car Cards: Top 3 recommended cars are shown as cards with essential specifications ("Price", "Model", "Mileage") and a direct link to the original listing in mobile.de.

This setup successfully provides chat-like search experience similar to AI search in mobile.de.

6 Sample search:



7 Conclusion

This capstone project successfully demonstrated replica for the advanced AI-Search feature developed by mobile.de on its website on local scale using open-source tools. Selenium for data extraction with a Hybrid RAG Architecture (Vector Search + Regex Filters + Cross-Encoders) have filled the gap between rigid parametric filters and the natural human intent. The final application allows users in Kiel to find cars using natural human language that fulfils the project's goal of developing AI based search.

8 References

- Retrieval Augmented Generation (RAG) in Azure AI Search. [1]
- Mobile.de used for scraping data [2]