Primjena velikih jezičnih modela u kreiranju personaliziranih planova putovanja temeljenih na činjenicama

Tajz, Sven

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:837071

Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija

Sveučilišni diplomski studij Informatika

Sven Tajz

Primjena velikih jezičnih modela u kreiranju personaliziranih planova putovanja temeljenih na činjenicama

Diplomski rad

Mentor: Prof. dr. sc. Sanda Martinčić-Ipšić

Rijeka, rujan, 2024.

Sveučilište u Rijeci
FIDIT Fakultet informatike i digitalnih tehnologija

Rijeka, 5.3.2024.

Zadatak za diplomski rad

Pristupnik: Sven Tajz

Naziv diplomskog rada: Primjena velikih jezičnih modela u kreiranju personaliziranih planova putovanja temeljenih na činjenicama

Naziv diplomskog rada na eng. jeziku: Personalized Travel Assistant Based on Retrieval Augmented Generation with Large Language Models

Sadržaj zadatka:

Razvojem velikih jezičnih modela u suvremene aplikacije uključuju se interakcija s korisnicima u prirodnom jeziku. Budući da je poznati problem generiranja teksta tzv. halucinacija koja se odlikuje koherentnim ali netočnim tekstom, potrebno je potražiti rješenja u generiranju tekstova temeljenih na činjenicama (Retrieval Augmented Generation).

U diplomskome radu će se razviti sustav za presonalizirano planiranje putovanja temeljeno na činjenicama zapisanih u tekstovima Wikitravel stranice. Razvijeno rješenje će se implementirati u web aplikaciji kroz korisničko sučelje u prirodnome jeziku. Tako će korisnik moći postavljati pitanja poput gdje se smjestiti, koje znamenitosti posjetiti, koje prijevozno sredstvo koristiti i slično, te dobivati točne i relevantne odgovore. U radu će se primijeniti odgovarajući postupci za segmentaciju i vektorizaciju teksta u niskodimenzionalnu reprezentaciju te postupci pretraživanja u vektorskoj bazi podataka. Sustav za personalizirano planiranje putovanja će se implementirati korištenjem Langchain python biblioteke.

Voditeljica za diplomske radove:

Prof. dr. sc. Ana Meštrović

Mentor:

Prof. dr. sc. Sanda Martinčić-Ipšić

Komentor:

Zadatak preuzet: 15.4.2024.

(potpis pristupnika)

Sažetak

Ovaj magistarski rad, istražuje razvoj i implementaciju inteligentnog asistenta dizajniranog za poboljšanje korisničkog iskustva putovanja kroz integraciju naprednih tehnika obrade prirodnog jezika. Osnovni fokus ovog istraživanja je korištenje „Retrieval augmented generation "(RAG) tehnike u kombinaciji s velikim jezičnim modelima (LLM) kako bi se u odgovor uključile točne informacije potkrijepljene na pronađenim činjenicama.

Rad započinje raspravom o ključnim konceptima koji su ključni za razumijevanje tehnologije koja stoji iza asistenta, uključujući velike jezične modele, problem halucinacije, RAG, vektorske ugradnje i vektorske baze podataka, također pruža sveobuhvatan pregled postojećih aplikacija koje koriste RAG ekstenzije.

U poglavlju o tehnologiji, rad detaljno opisuje komponente i alate korištene u projektu, kao što su TravelTalk, WikiTravel, LangChain, PineCone i FastAPI. Metodologija opisuje razvijen postupak korak-po-korak, počevši od prikupljanja podataka s interneta i pripreme podataka, umetanja i ažuriranja vektora u vektorsku bazu podataka, do povezivanja OpenAI API-ja s vektorskom bazom podataka za pohranu i pretraživanje činjenica, te konačno, razvoja i integracije API-ja s frontend aplikacijom.

Rezultati implementacije pokazuju učinkovitost personaliziranog putnog asistenta u pružanju korisnicima prilagođenih putnih preporuka i informacija. Rad zaključuje analizom rezultata projekta, potencijalnim poboljšanjima i budućim smjerovima za unaprjeđenje mogućnosti asistenta. Ovaj rad doprinosi području AI-pokretanih putnih asistenata prikazujući praktičnu primjenu LLM-a i RAG-a, nudeći uvide u tehnička rješenja potrebna za budući razvoj personaliziranih AI rješenja.

Ključne riječi: LLM, RAG, Hallucinations, Vector embedding, Vector Database, ChatGPT, Chunking, VueJS, NodeJS, Prompt engineering

Abstract

This master's thesis explores the development and implementation of an intelligent travel assistant designed to enhance user travel experiences through the integration of advanced natural language processing techniques. The core focus of this research is the utilization of Retrieval-Augmented Generation (RAG) technique combined with large language models (LLMs) to deliver accurate, contextually relevant information and recommendations.

The thesis begins by discussing key concepts crucial to understanding the technology behind the travel assistant, including large language models, the hallucination problem, RAG, vector embeddings, and vector databases. It provides a overview of existing applications that utilize RAG extensions currently.

In the technology section, the thesis elaborates the components and tools used in the proposed solution, such as TravelTalk, WikiTravel, LangChain, PineCone, and FastAPI. The methodology outlines the step-by-step process, starting from web scraping and data preparation, embedding and upserting (Pinecone terminology used for simultaneous inserting and updating) vectors into a vector database, to connect the OpenAI API to the vector database for retrieval, and finally, developing and integrating an API with a frontend application.

Results from the implementation demonstrate the effectiveness of the personalized travel assistant in providing users with tailored travel recommendations and information. The thesis concludes with an analysis of the project's outcomes, potential improvements, and future directions for enhancing the assistant's capabilities. This thesis contributes to the field of AI-driven travel assistance by showcasing the practical application of LLMs and RAG, offering valuable insights for future developments in personalized AI solutions.

Key words: LLM, RAG, Hallucinations, Vector embedding, Vector Database, ChatGPT, Chunking, VueJS, NodeJS, Prompt engineering

Table of contents

1. Introduction

The rapid evolution of artificial intelligence (AI) has significantly transformed various aspects of our daily lives, and the travel industry is no exception. Despite the abundance of travel-related information available online, travellers often face challenges in finding accurate, relevant, and personalized information to plan their trips effectively. Traditional travel assistants and search engines, while useful, frequently fall short in delivering personalized, context-aware recommendations that cater to individual preferences and needs. This gap in the market highlights the necessity for a more advanced, intuitive, and personalized travel assistant that can integrate vast amounts of data to provide precise and tailored travel advice.

The primary problem this thesis aims to address is the inefficiency and lack of personalization in current travel assistance tools. Users often spend considerable time and effort sifting through generic travel information, which may not always align with their unique requirements. This can lead to suboptimal travel experiences, frustration, and missed opportunities. To bridge this gap, the development of a sophisticated travel assistant that leverages the power of large language models (LLMs) and retrieval-augmented generation (RAG) is imperative [1].

Motivated by the potential of cutting-edge AI technologies to improve user experiences, this thesis introduces a web application that implements a personalized travel assistant. This application harnesses the capabilities of RAG combined with LLMs to deliver highly relevant and individualized travel recommendations. By integrating travel datasets and advanced AI algorithms, the application can comprehend and respond to complex travel queries with precision and context-awareness, thus enhancing the overall travel planning process.

The personalized travel assistant developed in this thesis not only aims to improve the efficiency of retrieving travel-related information but also strives to offer a more enjoyable and stress-free travel planning experience. By addressing the limitations of existing tools and leveraging the strengths of modern AI technologies, this application aspires to set a new standard in personalized travel assistance, ultimately enriching the way users plan and experience their journeys.

The thesis is organized as follows. Chapter 1 introduces the research topic. Chapter 2 discusses key concepts such as large language models, the hallucination problem, retrieval-augmented generation, vector embeddings, and vector databases. Chapter 3 provides an overview of existing applications for RAG extension. Chapter 4 outlines the technology used, including TravelTalk, WikiTravel, LangChain, PineCone, and FastAPI. Chapter 5 details the implementation, covering web scraping, embedding vectors, connecting APIs, developing an API, and prompt engineering. Chapter 6 presents the results, while Chapter 7 concludes the thesis. Finally, Chapters 8 to 12 include literature, lists of tables, illustrations, attachments, and additional contributions.

2. Key concepts

2.1. Large language models

Large Language Models (LLMs) are a sophisticated category of artificial intelligence (AI) models designed to understand and generate human language with remarkable proficiency [19]. These models are constructed using advanced deep learning techniques and are trained on extensive corpora of textual data. This expansive training allows LLMs to grasp the complexity of language, including grammatical rules, semantic meanings, and contextual cues. At the heart of many LLMs, such as GPT-3 (Generative Pre-trained Transformer 3), lies the Transformer architecture, which was originally introduced by Vaswani et al. in 2017 [7]. The Transformer model revolutionized natural language processing by employing self-attention mechanisms that enable the model to efficiently process and generate text. This self-attention mechanism allows the model to weigh the importance of different words in a sentence relative to each other, thereby producing sentences that are not only coherent but also contextually pertinent. The architecture's ability to handle long-range dependencies in text makes it exceptionally effective at capturing and replicating complex language patterns, leading to the generation of text that closely mirrors human communication. As a result, LLMs have become integral in various applications, ranging from chatbots to content creation tools, due to their advanced capability to mimic human-like responses and engage in meaningful interactions.

The training of LLMs comprises two fundamental stages: pre-training and fine-tuning [19]. The pre-training phase involves autoregressive training of the model on a vast corpus of text data - training the model on a task of predicting the next word. This phase is unsupervised, meaning that the model learns from patterns in the data without explicit human intervention or predefined labels [8]. During this stage, the model develops a broad and generalized understanding of language, including syntax, semantics, and various linguistic structures. It learns to recognize patterns, idiomatic expressions, and contextual clues that are common across diverse texts. The second stage, fine-tuning, involves a more targeted approach where the model is trained on a smaller, task-specific dataset [19]. Examples of tasks include text classification (e.g., spam detection, topic categorization), sentiment analysis (e.g., identifying positive or negative reviews, gauging social media sentiment), language translation (e.g., translating documents, converting speech to text across languages), and text summarization (e.g., summarizing news articles, generating abstracts for research papers). This stage is supervised, with human-provided labels guiding the model to refine its performance in a particular domain or application [8]. Fine-tuning helps the model adapt its generalized knowledge to specialized tasks, such as medical diagnosis or legal document analysis, enhancing its accuracy and relevance in those specific areas. The combination of these two stages ensures that LLMs can effectively generalize from a wide range of topics while also delivering high performance in specialized applications.

The remarkable effectiveness of LLMs in diverse applications such as language translation, text summarization, and conversational agents can be attributed to their ability to generate

human-like text based on the input they receive. By leveraging their extensive training data, LLMs can comprehend context, disambiguate meanings, and produce responses that are contextually appropriate and coherent. Their proficiency in generating natural language responses makes them invaluable tools in various fields, including customer service, content creation, and language education. However, despite their impressive capabilities, LLMs face several challenges. These include the significant computational resources required for their training and operation [8], as well as the potential for generating biased or inaccurate information [2]. These issues arise due to the biases present in the training data and the inherent limitations of the models. Addressing these challenges is a focal point of ongoing research, which aims to improve the reliability and efficiency of LLMs. Efforts are being made to develop more robust models that can minimize biases and enhance the accuracy of the generated information [10], ensuring that LLMs continue to evolve as powerful and dependable tools in the realm of artificial intelligence.

2.2. Hallucination Problem

The current definition of hallucinations [2], characterize them as generated content that is nonsensical or unfaithful to the provided source content. These hallucinations are further categorized into intrinsic hallucination and extrinsic hallucination types, depending on the contradiction with the source content. In a extensive research paper called „A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions “[2], hallucinations have been categorized into two distinct types. The first type called f**actual hallucination**; the generated response will provide answers that are contained to the context of the query, but the factual information is completely false. Whereas the second type called **faithfulness hallucinations** completely stray from the context of the user query.

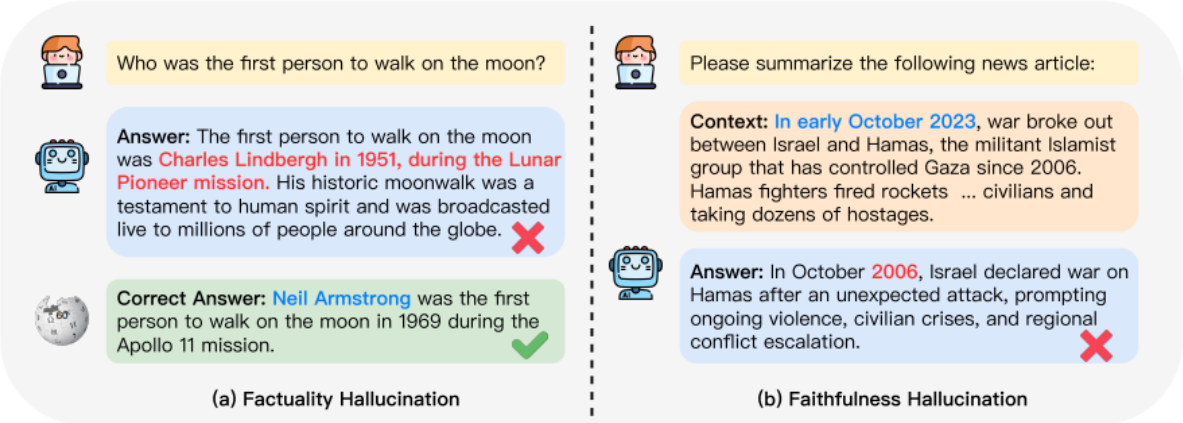


Figure 1Factual and faithfulness hallucination examples [2]

Hallucinations in LLMs can be attributed to several underlying factors. One primary cause is the quality of training data. LLMs like GPT-3 are trained on vast amounts of text data [8], which inherently contains inaccuracies, biases, and fictional content. During training, the model does not differentiate between factually accurate and inaccurate information, leading to potential misinformation generation. Additionally, the architecture of LLMs, primarily based on transformers, relies on patterns and associations in the training data to predict the next word

in a sequence. This pattern-based prediction, while powerful, can sometimes produce plausible but incorrect information if the context closely matches parts of the training data that contain errors. Furthermore, LLMs are designed to generalize from their training data, which, while enabling the generation of creative and contextually appropriate responses, also increases the risk of generating content that sounds correct but is not backed by verifiable facts.

Consider a scenario where an LLM is asked to provide information about a specific medical condition. If the model generates a plausible sounding but incorrect treatment method, it could lead to misinformation that might have serious consequences for individuals relying on that information. Similarly, in the legal domain, an LLM providing incorrect legal advice could result in significant legal repercussions.

Several strategies can also be employed to mitigate the number of occurrences of hallucinations without implementing Retrieval Augmented Generation [1]. One approach is enhancing the quality of the training data by curating and filtering for accuracy, which involves removing or tagging dubious sources and prioritizing verified information. Leveraging user feedback to identify and correct recurring hallucination patterns is also effective; users can flag incorrect information, which can then be used to fine-tune and improve the model [2]. Combining LLMs with other AI models specialized in fact-checking and data verification can create a more robust system that reduces the likelihood of hallucinations. Moreover, ensuring that users are aware of the potential for hallucinations and encouraging critical evaluation of the information provided by LLMs is essential.

2.3. Retrieval Augmented Generation

Retrieval-Augmented Generation (RAG) is an advanced natural language processing (NLP) technique that combines pre-trained, parametric-memory generation models with a non-parametric memory through a general-purpose fine-tuning approach [1] to produce more accurate and contextually relevant responses. RAG aims to address the limitations of purely generative models, which can sometimes produce inaccurate or nonsensical outputs (hallucinations) which will be explained later in more detail, by integrating information from external documents or databases.

RAG operates in two primary stages: retrieval and generation [1]. In the r**etrieval stage**, the input query is first processed to generate a search query. This could be a direct transformation of the user's input, or a more refined query aimed at fetching the most relevant information by embedding the query. The generated search query is then used to search a large corpus of documents. This corpus can be anything from a predefined vector database (examples: Pinecone, Qdrant, ElasticSearch, Redis etc.) to the search of the entire internet, depending on the use case. The retrieval mechanism typically employs dense or sparse retrieval methods. Dense retrieval utilizes embeddings from models like BERT which stands for Bidirectional Encoder Representations from Transformers [1] to find semantically similar documents, while sparse retrieval uses traditional search techniques like TF-IDF (Term Frequency-Inverse

Document Frequency) or BM25 to match keywords in the query with documents. Retrieved documents are then scored based on their relevance to the query, and the top-K documents with the highest scores are selected for the next stage.

In the **generation stage**, the selected documents are embedded using a transformer-based model to capture the contextual information. These embeddings are combined with the original query's embedding to form a comprehensive context vector. A generative model, often based on the Transformer architecture (e.g., GPT-3.5), uses this context vector to generate a coherent and contextually relevant response. This model is fine-tuned to ensure that it can effectively incorporate the information from the retrieved documents.

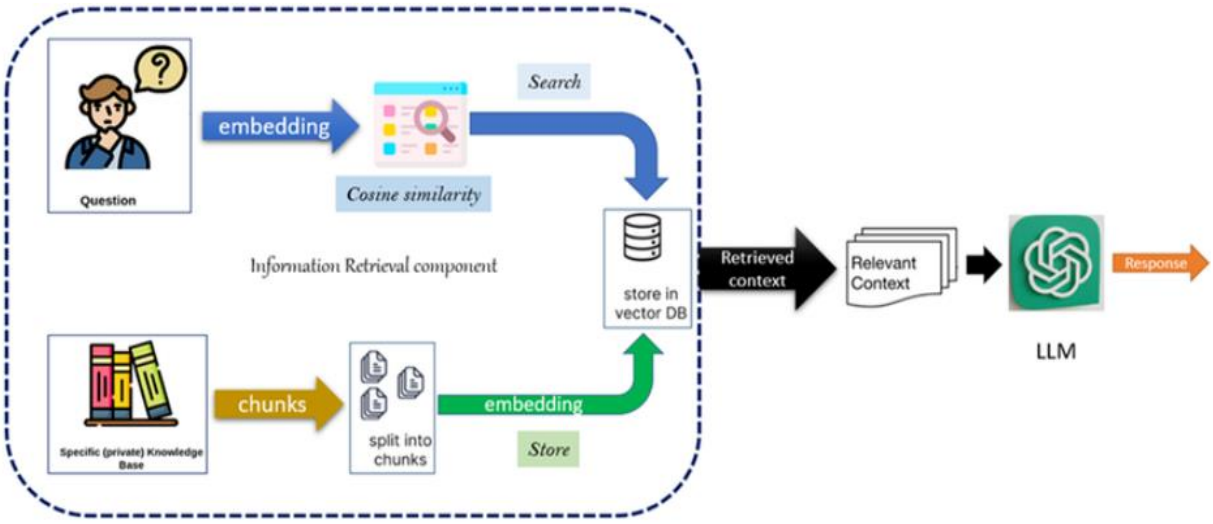


Figure 2 Visual demonstration of how RAG works

The benefits of RAG are numerous. By leveraging external knowledge, RAG is more strongly grounded in real factual knowledge which makes it “hallucinate” less and offers more control and interpretability [1]. RAG can be adapted to various domains by changing the corpus of documents used in the retrieval stage, making it a versatile tool for numerous applications, including personalized travel assistants like in this use case, customer support, educational tools, and simple question and answering assistants for specific events.

2.4. Vector embeddings

Embeddings are a fundamental component of neural language models, enabling these models to understand and generate human-like text [14]. By converting words, phrases, and sentences into dense, continuous vectors of numbers, embeddings bridge the gap between raw text data and the numeric representation that enables computational processes within the model. These vectors capture semantic meanings and relationships, allowing large language models (LLMS) to perform various natural language processing tasks effectively.

At the core of vector embeddings is the idea of projecting words from a high-dimensional sparse space to a low dimensional dense space where words with similar meanings are located closer to each other. This concept can be traced back to early models like Word2Vec and GloVe [12], which transformed individual words into fixed-size vectors based on their context within large corpuses. The training process for these embeddings involves predicting the semantic meaning of a word given its surrounding words or vice versa, leveraging the distributional hypothesis that words appearing in similar contexts tend to have similar meanings.

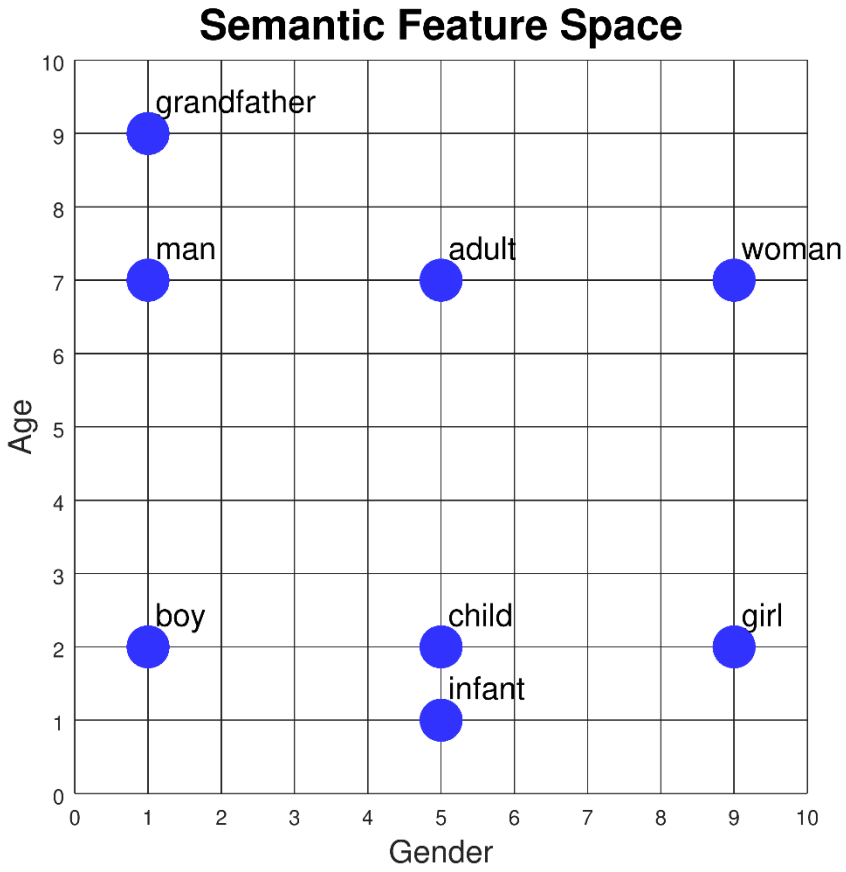


Figure 3 Visual representation of semantic meaning in a high dimensional space

Modern large language models, such as BERT and GPT-4, extend this idea by enabling the generation of more sophisticated embeddings that capture not only word-level semantics but also contextual information [14]. These models use transformer architectures, which apply attention mechanisms to weigh the importance of each word in a sentence relative to others. This allows the generation of contextual embeddings, where the representation of a word varies depending on its context. As depicted in Figure 3 the word “adult” semantically is between words “man” and “woman” while being far away from words such as “child” and “boy”.

Embeddings enable the model to understand nuanced differences between words and phrases, making it possible to generate more accurate and contextually appropriate responses [14]. For example, in a sentence completion task, embeddings help the model predict the most likely

next word based on the context provided by preceding words. But depending on the corpus that was used for vector embedding, the large language model can become biased when answering certain questions. Data is still the most important part of natural language processing, and machine learning in general. There is always risk when embedding corpuses that use certain words or sentences in different ways, which can then prompt the LLM to answer user questions in undesirable ways. Addressing this requires careful curation of training data and techniques such as debiasing algorithms to mitigate the impact of those biases [11].

2.5. Vector Database

A vector database is used to store high-dimensional data that relational database management systems cannot handle [14]. The primary function of vector databases is to enable fast and accurate similarity searches among vectors. When LLMs generate embeddings — dense, multi-dimensional vectors representing words, sentences, or documents — vector databases allow these embeddings to be stored and quickly retrieved based on their similarity to a query vector using an algorithm such as the similarity search algorithm [14]. This capability is crucial for tasks like semantic search, recommendation systems, and nearest neighbour searches, where finding the most similar items to a given input is required. Usually, these algorithms return the K most similar queries that are then ranked for further use.

However, the implementation and management of vector databases comes with challenges. One major challenge is the need for efficient storage solutions. High-dimensional vectors consume substantial storage space, and managing this data efficiently requires advanced compression techniques and scalable storage infrastructures [14]. Vector databases employ various storage techniques to manage and combat those challenges efficiently. Storage techniques include partitioning and sharding, compression algorithms and in-memory storage. Partitioning is referred to dividing a single database into segments that can be accessed independently, while sharding refers to distributing those partitions on different machines for load balancing [14]. In-memory storage techniques store certain vector data in random access memory instead of a disk for faster search and retrieval [14].

The use of vector databases greatly enhances the performance of LLMs across various applications. In semantic search, for example, vector databases enable users to find documents or text that are semantically similar to a query, even if they do not share the same keywords. This results in more relevant search outcomes that better align with the user's intent. This capability is implemented through advanced techniques such as Locality-Sensitive Hashing (LSH), which hashes input items so that similar items map to the same buckets with high probability, significantly reducing the search space [14]. Additionally, tree-based indexing methods like k-d trees and ball trees create a hierarchical structure to enable efficient range and nearest neighbour queries by recursively partitioning the data space [14].

In recommendation systems, vector databases can store user preferences and item characteristics as vectors, allowing the system to suggest items that closely match a user's past interactions or stated preferences. This process is often facilitated by graph-based indexing methods such as Hierarchical Navigable Small World (HNSW) graphs as illustrated in Figure 4, which represent vectors as nodes and establish edges based on proximity, enabling efficient traversal during queries [13].

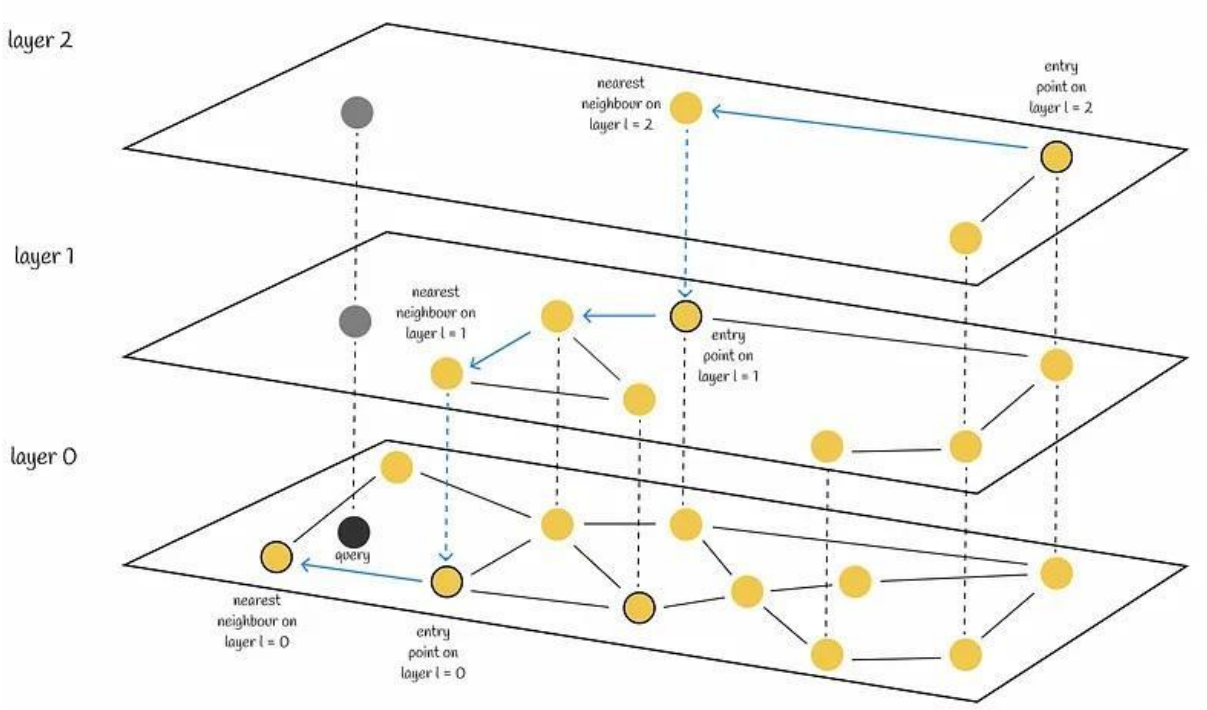


Figure 4 Hierarchical Navigable Small World Graph [38]

Moreover, vector databases support the scalability of LLM applications. As the volume of data and number of embeddings increase, vector databases can manage the additional load without significant performance loss. This scalability is crucial for maintaining the efficiency and responsiveness of LLM-based systems, especially in real-time applications like chatbots, virtual assistants, and personalized content delivery. The combination of advanced indexing techniques ensures that vector databases can handle high-dimensional data efficiently, providing robust and scalable solutions for a variety of applications.

While approximate nearest neighbour algorithms can speed up searches, there is often a trade-off between accuracy and speed, which must be carefully balanced depending on the application's requirements. Security and privacy are also significant concerns [13]. Vector databases can store embeddings that contain sensitive information, and unauthorized access to this data can lead to privacy breaches. Implementing robust security measures, such as encryption and access controls, is essential to protect the data stored in vector databases. Additionally, ensuring that the vectors do not inadvertently encode private or sensitive information through techniques like differential privacy can further enhance data security [14].

Another challenge is the integration of vector databases with existing data storage and processing systems and workflows. Many organizations have legacy systems designed around relational databases, and integrating vector databases requires careful planning and potentially significant changes to data pipelines and architectures. Interoperability and seamless integration are crucial for leveraging the full benefits of vector databases without disrupting existing operations.

Vector databases represent a critical technology for advancing the capabilities of LLMs and other AI systems. By efficiently storing and retrieving high-dimensional data, they enable a wide range of applications, from semantic search to recommendation systems. Addressing the challenges of scalability, latency, security, and integration is essential for the successful deployment of these databases [14]. As research and development continue, vector databases will play an increasingly important role in the data-driven landscape.

3. Overview of Applications for RAG Extension

Retrieval-Augmented Generation (RAG) as a technique has been adopted by various companies and applications to enhance the quality and relevance of generated content. RAG combines retrieval mechanisms with generative models, allowing systems to pull in pertinent information from a large corpus and use it to create more accurate and contextually appropriate responses [1]. This approach is increasingly being used across different domains and use cases to improve user interactions, content creation, and information synthesis.

Google employs RAG in its Question Answering (QA) systems to deliver more precise and informative responses by retrieving pertinent documents from extensive corpora [32]. Facebook RAG model, designed for open-domain QA, uses Dense Passage Retrieval (DPR) and BART to significantly improve answer quality and contextual relevance [1]. Zendesk enhances its customer support chatbots with RAG-based models, retrieving relevant knowledge base articles to provide instant, tailored responses, thereby reducing reliance on human agents and improving customer satisfaction [33]. IBM Watson's virtual assistants use similar techniques to deliver detailed, contextually appropriate responses, enhancing both customer satisfaction and support efficiency [34]. Copy.ai utilizes RAG to help marketers and writers generate high-quality content by retrieving relevant data from large corpora, producing well-informed and contextually accurate marketing materials [35].

Retrieval-Augmented Generation is revolutionizing the way various industries handle information and LLM-enabled content generation. By integrating retrieval and generative capabilities, RAG-based models are enhancing the factuality, quality, relevance, and efficiency of responses and content. This technology is proving to be invaluable in wide spectrum of applications ranging from customer support and virtual assistance to academic research and content creation.

4. Technology used

4.1. TravelTalk

TravelTalk is a single-page web application where users can post information about their travels and experiences. Every single post is geolocated using a location marker which is then displayed on a map, this makes it easier to share and write about experiences with other users. The application is still in active development and is the application into which the personalized travel agent extended with RAG is implemented. The application is split into two main parts, the client component, and the server component. The client component hosts the frontend of the application (as seen on Figure 5), which is being built using VueJS, a popular component-based JavaScript framework for building scalable and performant user interfaces for web applications [20]. The server component is further split into two interconnected parts. The REST API server, and the database. The database that is being used for this project is MongoDB [31]. MongoDB is a NoSQL (not only SQL) database that manipulates data by saving the data into documents instead of tables. The database saves records of user's information for logging into the application, their posts and interactions while the REST server communicates with the database and sends the responses to the frontend for easier use and visualization. Most of the workload for the application is distributed to the server component. The REST API server is being built using NodeJS, a popular JavaScript runtime that enables JavaScript to run outside of the browser using Google's V8 engine [21]. On the server component all error messages are being processed and handled to ensure security, authorization and authentication. If error handling, authentication or authorization were being handled in any capacity in the client component, it would be very prone to security hazards. Tech savvy users can manipulate the client-side code to enter unauthorized domains or API requests using techniques like operating the browsers local storage or intercepting and parsing cookies that are saved in local memory.

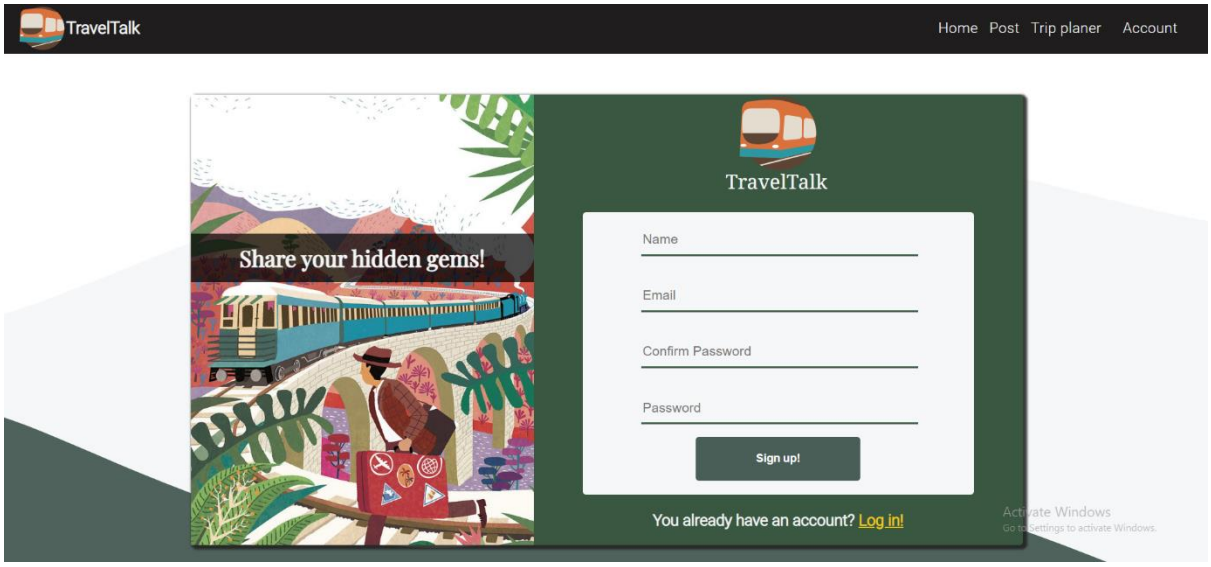


Figure 5 Visual interface of TravelTalk application

Retrieval augmented generation is implemented into the frontend of the TravelTalk application using an external and custom API. The final implementation will include a text box that will support a chatting system between the user and the RAG extended large language model.

4.2. WikiTravel

WikiTravel, a collaborative project launched in 2003, aims to create a comprehensive, up-to-date, and reliable free travel guide [22]. Operating as a wiki, it allows anyone with internet access to contribute, edit, and update its content. WikiTravel in the central point of access to the collective knowledge and experiences of travelers worldwide, resulting in a dynamic and continuously evolving resource. One of WikiTravel's primary strengths is its extensive and detailed information on a wide array of destinations, from popular tourist destinations to lesser-known locations. It offers insights into attractions, accommodations, dining, transportation, and local customs. Each location page typically includes sections on history, climate, geography, culture, and travel tips, ensuring a holistic view that caters to various traveler needs. This breadth of information makes WikiTravel an invaluable tool for travelers seeking authentic, practical, and diverse perspectives on their destinations. Additionally, it serves as an excellent dataset for knowledge extension for personalized travel assistants, hosting valuable and factual information that large language models may not possess in their training datasets. This includes contact details, addresses, menu prices for restaurants and establishments, shortcuts documented by local residents, and tips on security and navigating cities. The platform's open-editing model ensures a constant supply of up-to-date knowledge, further enhancing its utility for extending the capabilities of language models. All information on WikiTravel is open-sourced and well-documented, with detailed API documentation available on the MediaWiki platform [22]. MediaWiki, being open-source and continually supported, allows users to access and create custom applications for any website hosted using its engine, with Wikipedia and WikiTravel being prime examples.

WikiTravel’s well-organized structure enhances its usability, categorizing content into continents, countries, regions, cities, and various travel topics. This geographical organization, combined with hyperlinked entries to related articles, creates an interconnected web of information that is both comprehensive and easily accessible. This layout helps travellers plan their trips efficiently and serves as a resource for researchers and developers seeking detailed and structured information on global travel destinations. The thorough compilation and cross-referencing of data ensure users can find specific information quickly and efficiently, providing a wide range of interconnected information. This makes WikiTravel a practical tool for travellers and an exemplary resource for anyone involved in travel research or development. In conclusion, Wikitravel's collaborative nature, comprehensive information, and organized structure make it an indispensable tool for travellers, researchers, and developers alike, offering a reliable and expansive travel guide that continuously evolves with contributions from a global community.

4.3. LangChain

LangChain is an advanced Python framework designed to streamline the development and deployment of applications that utilize large language models (LLMs) [23]. Launched to meet the growing demand for development of LLM-enabled sophisticated natural language processing (NLP) applications, LangChain provides the necessary tools and infrastructure to facilitate seamless integration of LLMs into various workflows. By emphasizing modularity, extensibility, and ease of use, LangChain caters to a wide range of applications, from chatbots and virtual assistants to content generation and complex data analysis. This flexibility allows developers to create tailored solutions that meet specific requirements, making LangChain a versatile tool in the realm of NLP.

One of the primary strengths of LangChain is its modular architecture, which offers a high degree of customization. The framework is designed to let developers select and combine different components, known as "chains," according to their specific needs. This modularity spans various aspects of NLP [24], including tokenization, model selection, and text generation, enabling developers to experiment with different configurations and optimize their applications for performance and accuracy. In the context of this thesis, LangChain is particularly valued for its built-in support for the PineCone vector database [24], OpenAI embedding functionality [24], and chunking capabilities, especially through the Recursive Character Text Splitter [15]. Widely adopted, LangChain's detailed and comprehensible documentation, along with its “out of the box” code snippets, provides developers with a solid foundation for utilizing its modules effectively. This accessibility not only helps in rapid prototyping but also in fine-tuning applications to better align with unique project requirements.

Moreover, LangChain's user-friendly API abstracts much of the complexity involved in working with LLMs, making it accessible to developers with varying levels of expertise [24]. The framework includes pre-built components and templates for common tasks such as question answering, text summarization, and sentiment analysis, which allows for the quick building and deployment of functional NLP applications. For example, a Retrieval-Augmented Generation (RAG) application can leverage these pre-built components to analyse user queries, extract relevant information from sources like WikiTravel data, and generate personalized travel advice. This ease of use is complemented by LangChain's support for extensibility, allowing developers to enhance and expand the framework’s capabilities as needed. Custom components can be seamlessly integrated, facilitating the creation of specialized NLP functionalities. This extensibility is crucial for applications that require domain-specific knowledge or custom processing pipelines [24]. For instance, developers can create custom embeddings or fine-tune models to improve the relevance and accuracy of generations of recommendations, significantly enhancing the user experience.

LangChain's emphasis on performance and scalability is another critical aspect of the framework. It is designed to handle large volumes of data and high request rates, making it suitable for enterprise-grade applications; currently over one hundred thousand companies use LangChain for NLP related tasks [23]. The framework supports distributed processing and can

be deployed in various environments, including cloud platforms and on-premises infrastructures, ensuring that applications can scale to accommodate growing user bases and increasing data loads without compromising performance. Additionally, LangChain provides robust tools for monitoring and managing NLP applications, including features for logging, error handling, and performance tracking [24]. These tools are essential for identifying bottlenecks, diagnosing issues, and maintaining optimal performance in production environments. LangChain's focus on security and compliance further enhances its suitability for enterprise applications, with features for data encryption, access control, and adherence to data protection regulations [24]. This ensures that sensitive information is handled securely, which is particularly important for applications processing personal data or operating in regulated industries. LangChain’s combination of modularity, ease of use, performance, and security makes it a trending tool for developers building advanced LLM-enabled NLP applications.

4.4.PineCone

Pinecone is a vector database designed to streamline the storage, indexing, and retrieval of high-dimensional vector embeddings [25]. Developed to address the increasing demand for efficient management of vector data, Pinecone offers a managed infrastructure that simplifies the complexities associated with vector databases. This allows developers to concentrate on building and scaling their applications rather than focusing on data handling. By providing a robust and user-friendly platform (as seen on Figure 6), Pinecone enables s developers to leverage the full potential of vector embeddings for various applications, including similarity search, recommendation systems, and natural language processing [25].

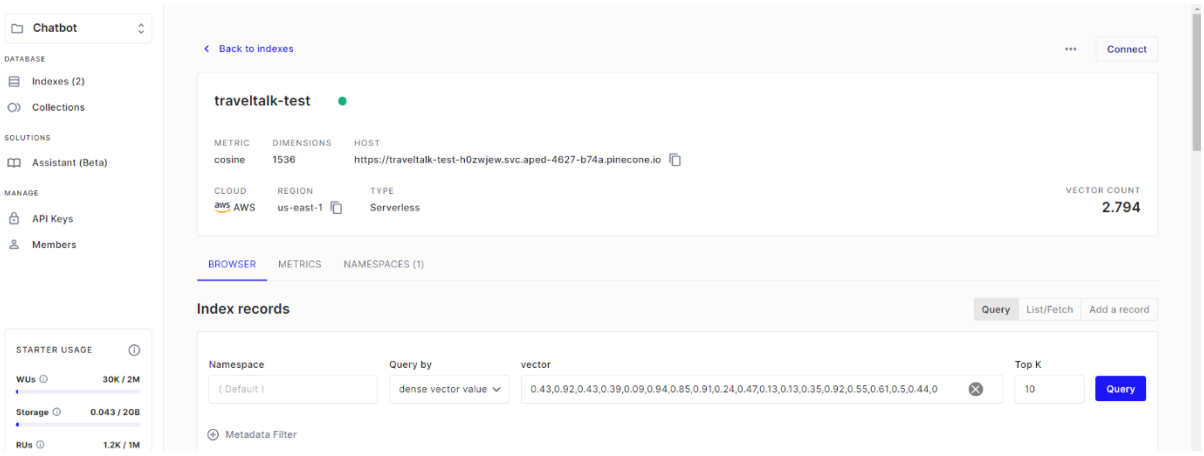


Figure 6 Figure 6 Pinecone user interface

A primary advantage of Pinecone lies in its capacity to manage and query vector embeddings at scale while maintaining high performance [25]. Vector embeddings, are essential for tasks that require retrieval and processing large amounts of data. Pinecone’s architecture is optimized to handle vector embeddings efficiently, offering fast and accurate similarity searches through

advanced indexing techniques such as hierarchical navigable small world graphs (HNSW) [25]. This capability is particularly beneficial for applications that need to process complex, high-dimensional data quickly and accurately. For instance, in the development of an application that uses Retrieval generated augmentation (RAG) using WikiTravel data, Pinecone can store and retrieve relevant vector embeddings, enabling the application to deliver precise and accurate travel advice and recommendations. When a user queries the RAG application, Pinecone swiftly retrieves the most pertinent vector embeddings, ensuring prompt and accurate responses, thereby enhancing the overall user experience.

Pinecone's fully managed service offers a range of features that significantly enhance the development process [25]. One of the key features is automatic scaling, which ensures that the system can handle increasing volumes of data and queries without any degradation in performance. This is crucial for maintaining efficiency as the application grows. Additionally, Pinecone includes built-in monitoring and logging capabilities, allowing developers to track usage patterns, identify potential issues, and optimize their applications accordingly [25]. This managed approach reduces operational overhead, freeing researchers and developers to focus on data utilization rather than infrastructure management. Furthermore, Pinecone supports integration with popular machine learning frameworks and libraries, making it easier to ingest and process vector embeddings from various sources [25]. This interoperability enhances its utility in diverse research and development contexts. For example, embeddings generated from WikiTravel data using models like BERT [36] or in this case OpenAI's embedding model [25], can be stored and queried within Pinecone, enabling data analysis and application functionalities. Pinecone's security features, including data encryption at rest and in transit [37], ensure that sensitive information is protected, providing assurance that user data and proprietary models are handled securely throughout their lifecycle.

Chunking helps mitigate these issues by breaking the corpus into manageable pieces, ensuring faster and more precise retrieval of information. By embedding smaller chunks, the vector database can quickly and accurately match queries with relevant data. This approach optimizes both the speed and reliability of the travel assistant, providing users with timely and accurate information even under heavy load.

Pinecone is a vector database tool for managing vector embeddings, offering unparalleled performance, scalability, and ease of use. Its ability to efficiently store and query vector data makes it a valuable asset for applications relying on complex data representations, such as the RAG application developed using WikiTravel data. By integrating Pinecone, developers can leverage its advanced features to build robust, responsive, and scalable systems that significantly enhance user experience and drive innovative solutions. Pinecone's combination of high performance, seamless scalability, and strong security measures makes it an ideal choice for any developer looking to harness the power of vector embeddings in their applications.

4.5. FastAPI

FastAPI is a cutting-edge, high-performance web framework for building APIs with Python 3.6+ that relies on standard Python type hints [26]. Created by Sebastián Ramírez, FastAPI is designed for ease of use and learning while maintaining efficiency and scalability. One of its standout features is the automatic generation of interactive API documentation using Swagger UI and ReDoc, making it a valuable tool for both developers and API consumers.

A significant advantage of FastAPI is its impressive performance. Leveraging the asynchronous capabilities of Python's asyncio module and the Starlette framework [17], FastAPI achieves exceptional speed. Benchmarks ran by independent company TechEmpower [16] indicate that FastAPI can rival frameworks like Express.js [16] in terms of performance, positioning it among the fastest frameworks available for Python. This performance is crucial for applications that demand low latency and high throughput, such as real-time data processing and high-concurrency services.

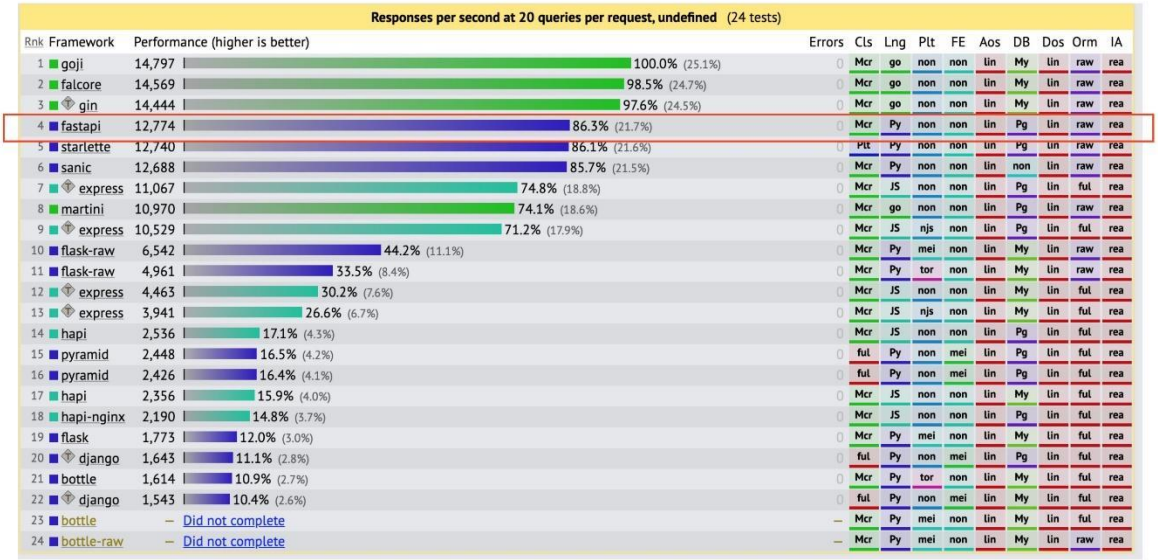


Figure 7 API performance benchmark

The ease of use provided by FastAPI is another advantage. Following modern Python practices, it is intuitive for developers familiar with the language. The use of Python type hints for request validation and dependency injection allows developers to write clean, readable code. This not only accelerates the development process but also makes the codebase easier to maintain and extend. Defining request models with Pydantic [18], for example, ensures automatic data validation, reducing boilerplate code and minimizing potential bugs. Additionally, FastAPI's support for asynchronous programming enables it to handle multiple requests concurrently without blocking, a feature critical for applications that process multiple user requests simultaneously. This is particularly beneficial for a RAG application serving real-time travel

recommendations, as FastAPI can efficiently manage I/O-bound operations, leading to better resource utilization and enhanced responsiveness.

FastAPI integrates seamlessly with other Python libraries and frameworks. It supports dependency injection [26], which allows for modular and reusable components, making it suitable for large-scale applications. Integrating FastAPI with machine learning libraries like TensorFlow or PyTorch [30], for instance, enables the creation of APIs that can serve ML models, enhancing the functionality of applications like the RAG system. The framework's auto-generated interactive documentation is another feature. This documentation is automatically updated as the API evolves, providing a clear and up-to-date reference for developers working on the project and for users needing to understand how to interact with the API. In collaborative environments, this ensures all stakeholders have access to accurate and comprehensive API documentation.

FastAPI places a strong emphasis on security, addressing common security requirements such as OAuth2, JWT (JSON web token) authentication, and API key validation [27]. This focus on security ensures that APIs built with FastAPI can protect sensitive data and provide secure access controls, which is critical for applications handling personal or proprietary information. By prioritizing security, FastAPI makes it possible to build robust, secure APIs capable of supporting a wide range of applications, from simple web services to complex, high-performance systems.

5. Implementation

5.1. Web scraping and data preparation

The first and most crucial step is acquiring and preparing the data for embedding and storage into the vector database. The goal of the travel assistant is to provide as much useful information as possible about various cities and popular travel destinations. Due to the extensive data required to cover the entire globe, it was decided to include data only from the 50 most popular tourist destinations in Europe [31]. This dataset is extensive to provide the necessary information without being computationally expensive to process and embed. Additionally, limiting to Europe, enabled detection of possible inaccuracies or hallucinations, due to the personal familiarity with some of the locations.

If there is a need to extend the data to include other cities or continents, the system and framework have been developed to allow easy addition or removal of information. All information was sourced from WikiTravel. The website contains information such as where to eat when visiting, various ways to navigate to and from a selected destination, cultural details, and much more. All of the data can be accessed from the MediaWiki platform on which WikiTravel is hosted. To access the data, an external API call needs to be made with a specific query indicating the desired data. Figure 8 provides a visual representation of MediaWiki's documentation, displaying the available HTTP properties when requesting data and the responses the server could return.

GET request [edit]

api.php ? action=parse & page=Pet_door & format=json [try in ApiSandbox]

Response [edit]


```
{
    "parse": {
        "title": "Pet door",
        "pageid": 3276454,
        "revid": 852892138,
        "text": {
            "*": "<div class=\"mw-parser-output\"><div class=\"thumb tright\"><div class=\"thumbinner\"
style=\"width:222px;\"><a href=\"/wiki/File:Doggy_door_exit.JPG\" class=\"image\"><img alt=\"\"
src=\"//upload.wikimedia.org/wikipedia/commons/thumb/7/71/Doggy_door_exit.JPG/220px-Doggy_door_exit.JPG\" width=\"220\"
height=\"165\" class=\"thumbimage\"
srcset=\"//upload.wikimedia.org/wikipedia/commons/thumb/7/71/Doggy_door_exit.JPG/330px-Doggy_door_exit.JPG 1.5x,
            ...
        }
    }
}
```


Figure 8MediaWiki Get request

For this specific use case the following URL was used:

```
base_api_url =
"https://wikitravel.org/wiki/en/api.php?action=parse&prop=wikitext&page={city}&format=json"
```

To access the WikiTravel data using an API call, the base URL needs to be "**wikitravel.org/wiki/en/api.php**". Each property sent with the request is separated by the "&" symbol. In this case, the properties are "action=parse", "prop=wikitext", "page={city}", and "format=json". This request retrieves all the data about a certain city in WikiTravel's database and parses the data in JSON format. With this step, we have only acquired access to the database; we still need to build our corpus by fetching, cleaning, and storing the data. The full function for accessing and saving data can be seen below:

```
def fetch_and_save(api_url, city):
    try:
        headers = {
            'User-Agent': '{LOGIN_TOKEN}'
        }

        response = requests.get(api_url, headers=headers)


        if response.status_code == 200:
            try:
                content = response.json()
                content_str = json.dumps(content, indent=4,ensure_ascii=False)
                filename = f"markdowns/{city.replace(' ', '_')}.md"

                with open(filename, 'w', encoding='utf-8') as file:
                    file.write(content_str)

                print(f"Response saved to {filename}")
            except KeyError:
                print(f"No data found for {city}")
        else:
            print(f"Error: {response.status_code}")

    except requests.RequestException as e:
        print(f"Error fetching data: {e}")
```

The function itself takes two input variables: the base URL ("https://wikitravel.org/wiki/en/api.php?action=parse&prop=wikitext&page={city}&format=json") and the city. The function sends a GET request to the MediaWiki platform, and if the response status code is 200 (indicating success), the content is written to a markdown file and saved to a directory called "markdowns". If any of these steps fail, an error will be generated.

Since all the data is in JSON format with many HTML tags and formatting, the data needs to be prepared and cleaned for future embedding. A function that removes all whitespaces, HTML

tags, and newlines and sets all characters to lowercase (to help with retrieval later on) was written, as seen below:

```
def preprocess_text(text):
    text = re.sub(r'\n+', ' ', text)
    text = re.sub(r'[\[\]\{\};.,=<>]', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    text = text.lower()
    return text

def clean_files(directory):
    for filename in os.listdir(directory):
        if filename.endswith(".md"):
            filepath = os.path.join(directory, filename)
            with open(filepath, 'r+', encoding='utf-8') as file:
                text = file.read()
                cleaned_text = preprocess_text(text)
                file.seek(0)
                file.write(cleaned_text)
                file.truncate()
```

With these crucial steps completed, the corpus is successfully prepared and pre-processed for further use in the next steps. This comprehensive preparation involves not only accessing the raw data from the WikiTravel database but also ensuring that the data is cleaned and standardized for optimal performance. By removing HTML tags, whitespaces, and formatting inconsistencies, we create a streamlined and consistent dataset. Additionally, converting all text to lowercase improves the efficiency of future data retrieval and embedding processes. This preprocessing sets a solid foundation, enabling the travel assistant to deliver accurate and relevant information efficiently. The corpus is now ready for advanced stages, including embedding into the vector database and further analysis.

5.2. Embedding and upserting vectors into a vector database?

After the data preparation step is complete; the next step is to embed the finalized corpus and insert all the data into a vector database. For this thesis, the Pinecone vector database was chosen, but the key steps and concepts apply to any vector database.

Since the WikiTravel corpus is quite large, containing around three hundred thousand words, we first need to split all the files into smaller chunks. This method, known as "chunking," is essential for efficient processing [28]. If we were to embed and insert all fifty files into the database as is, querying the database for specific information would require the engine to process entire files before locating the required information. This would dramatically decrease performance and accuracy, especially at scale when thousands of users are querying the database simultaneously.

The function “chunk_data” as seen below reads the input file, and splits the file into smaller chunks, where the chunk size is set to two thousand characters, and every subsequent chunk overlaps with one hundred fifty characters, and then returns the list of chunks into an array.

```
def chunk_data(filename, chunk_size=2000, chunk_overlap=150):
    text = filename

    text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size,
chunk_overlap=chunk_overlap, length_function=len, is_separator_regex=False)
    chunks = text_splitter.split_text(text)

    return chunks
```

The trick here is to flatten the final list to prevent nested arrays which can break the process of inserting data to the database. Once the files have been successfully chunked and saved as an array, an embedding model needs to process all of the chunks. For the purpose of this thesis, using Langchains OpenAI embedding API all of the chunks from all WikiTravel files can be embedded into the vectors and stored in the vector database.

```
embeddings = OpenAIEmbeddings(api_key=os.environ['OPENAI_API_KEY'])
index_name='traveltalk-test'

vector_store = PineconeVectorStore.from_documents(
        final,
        index_name=index_name,
        embedding=embeddings
    )
```

The process of inserting data is quite simple, using Pinecone’s built-in (“from_documents”) function we can select the documents we want to embed and insert. The variable called “final” represents the document list, index_name represents into which index or collection we want to insert the data and the embeddings variable specifies the embedding model the function will use.

Once this step completes all of the data can be seen in the Pinecone dashboard. The current number of vectors is 2794 as seen on Figure 9.

Figure 9 TrevelTalk vector count in Pinecone dashboard

5.3. Connecting OpenAI API to Vector database for retrieval

OpenAI's API is locked behind a paywall, but currently, as of writing this thesis, the price is almost negligible and provides plenty of features. The main feature being used for this thesis is the ability to pick and use a specific generative AI model. For the purpose of this thesis, the "gpt-3.5-turbo" model was picked due to its stability and relatively consistent performance. Once the API paywall is handled, we receive an API key that we can use for our RAG-extended travel assistant. Importing and initializing LangChain's OpenAI chat module is fairly simple. Here is the code snippet to do so:

```
from langchain_openai import ChatOpenAI

chat = ChatOpenAI(
    openai_api_key = os.environ['OPENAI_API_KEY'],
    model = 'gpt-3.5-turbo'
)
```

LangChain provides a robust interface for integrating with OpenAI's API, making it easier to build applications that leverage the power of large language models. With this setup, we can interact with the selected AI model to enhance our travel assistant, ensuring it can provide accurate information to users. Once the chat is initialized, we need to create an augmented prompt that includes insertion of the retrieved knowledge from vector database. When creating an LLM-assistant, first we need a system prompt that instructs the generative model on how to behave, what questions it can answer, what the user query is, and what knowledge the model has to answer the query, providing the context of the interaction.

We can create a simplified function to test if our retrieval method works correctly. This function, called "augmented_prompt," can be seen in the following code snippet:

```
def augmented_prompt(query: str):
    results = vectorStore.similarity_search(query,k=3)
    source_knowledge = '\n'.join([x.page_content for x in results])

    augmented_prompt = f"""Using the contexts below, answer the query.

    Contexts: {source_knowledge}

    Query: {query}"""
    return augmented_prompt
```

In this function, the similaritysearch method from Pinecone's vectorStore method is used to retrieve the top three most relevant pieces of information related to the query. These pieces of information are then combined into a single string, "source_knowledge", which is incorporated into the prompt. The prompt instructs the model to use the provided contexts to generate a response to the query.

Writing effective system prompts is crucial for ensuring that the generative model behaves as expected and provides accurate and relevant answers. By embedding the retrieved knowledge directly into the prompt, we enable the model to leverage specific information from our database, improving its ability to assist users with their inquiries. This approach helps create a more robust and informative AI assistant, with reduced problem of the hallucinating while generating response.

5.4. Test

Now we need to test implemented function and confirm if the retrieval of data works. For this experiment, we will use an identical query on both our extended assistant and the official ChatGPT platform. Both tests will use the same model for consistency. Below, you can see the responses from the RAG extended model and the ChatGPT model, where we send the exact same query. In this case, the query is, *"What's the phone number of Parish of the Blessed Trinity in Amsterdam*?".

In this experiment, we aim to evaluate the performance and accuracy of our retrieval-augmented generation (RAG) extended model compared to the standard ChatGPT model. By using the same query across both platforms, we ensure a fair comparison and can identify any discrepancies in the responses. This will help us determine if our extended model improves information retrieval and overall user experience. The specific query chosen for this test is practical and straightforward, making it an ideal candidate for assessing the models' efficiency and reliability in retrieving real-world data.

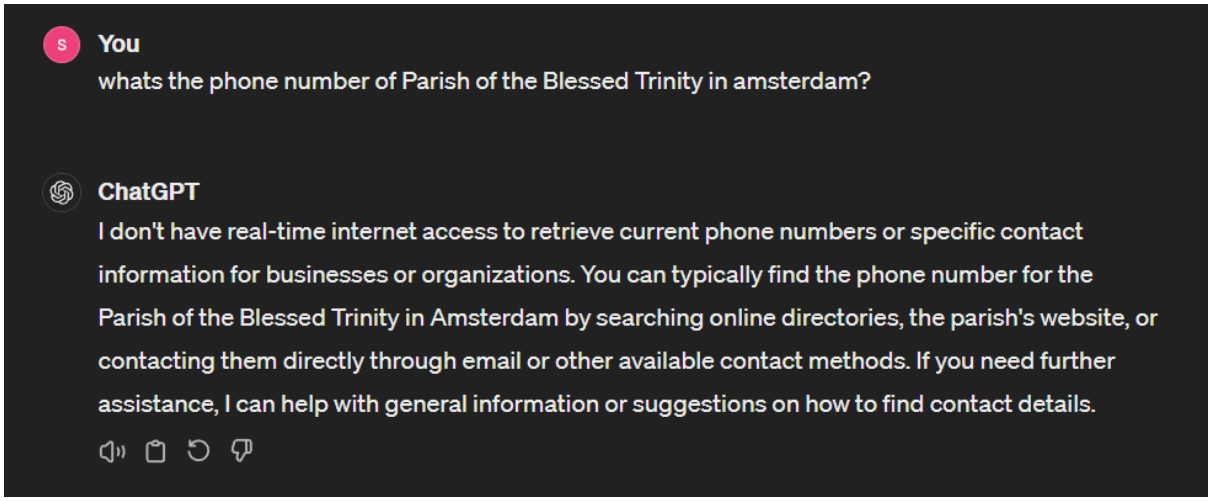


Figure 10 ChatGPT response

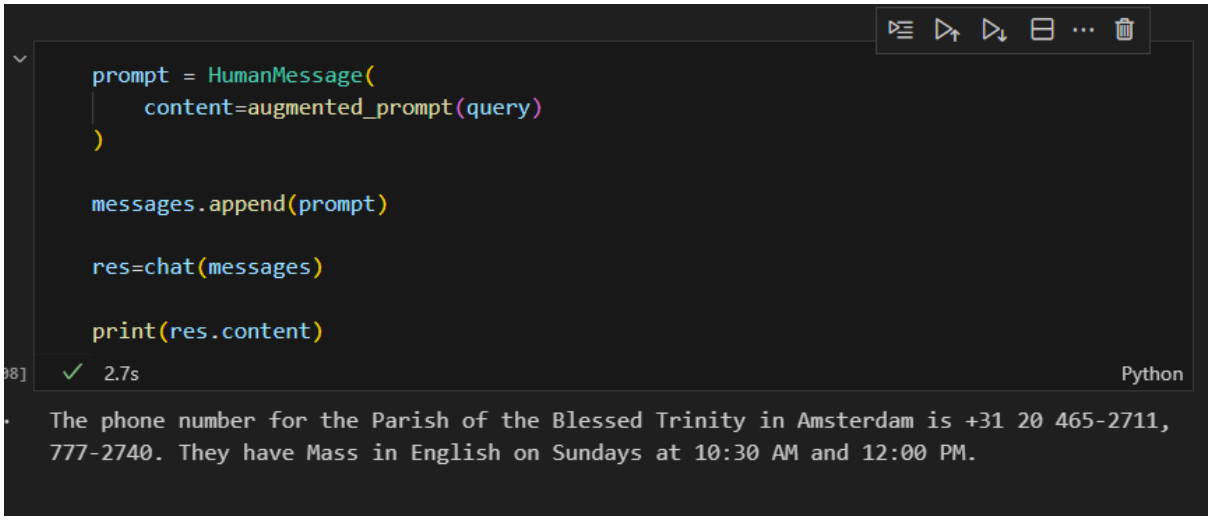


Figure 11 Rag extended model responseWe can now confirm the RAG extended model works as intended and retrieves correct information, overperforming the ChatGPT-turbo-3.5 response.

5.5. Developing an API

To utilize and connect the RAG (Retrieval-Augmented Generation) extended travel assistant, it is imperative to create an API that facilitates communication between the frontend application, TravelTalk, and the assistant, enabling the display of messages. The development of this API employs FastAPI.To allow the frontend application to communicate with the API, Cross-Origin Resource Sharing (CORS) must be configured. This setup specifies the origin is „localhost:8080“(which is the address of the TravelTalk application) to allow access to the API, thus enhancing security while enabling necessary functionality:

```
origins = ["http://localhost:8080", "http://127.0.0.1:8080"]
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Next, we need to initialize the Pinecone client and the OpenAI chat model using API keys stored in environment variables. Defining the embeddings and vector store which will be used to retrieve similar contexts for the given queries:

```
class Query(BaseModel):
    query: str

pc = Pinecone(api_key=os.environ['PINECONE_API_KEY'])
chat = ChatOpenAI(
```

```
    openai_api_key=os.environ['OPENAI_API_KEY'],
    model='gpt-3.5-turbo'
)

embeddings = OpenAIEmbeddings(api_key=os.environ['OPENAI_API_KEY'])
indexUpsert = pc.Index('traveltalk-test')
text_field = 'text'
vectorStore = PineconeVectorStore(
    indexUpsert,
    text_key=text_field,
    embedding=embeddings,
)
```

Creating a function „augmented_prompt“ that generates a comprehensive prompt by combining retrieved contexts with the user query. This function uses the vector store to perform a similarity search and constructs a detailed prompt for the assistant to provide accurate and contextually relevant responses:

```
messages = []

def augmented_prompt(query: str):
    results = vectorStore.similarity_search(query,k=5)
    source_knowledge = '\n'.join([x.page_content for x in results])

    augmented_prompt = f"""  You are a helpful and friendly travel assistant.
    Using 10th grade knowledge, you will answer user queries about anything
related to traveling. Questions
    about cities, navigation, cuisine etc.
    If a question arises about a different area, respectfully decline the
query.
    Using only the knowledge from the contexts below, answer the query.
    Here is the conversation history: {messages}
    Contexts: {source_knowledge}

    Query: {query}"""
    return augmented_prompt
```

The function uses two main variables for retrieval. The “messages” variable is an array structure that holds each message sent by the LLM and the user sending queries. Using this variable in the prompt we can access the conversation history for better contextual performance. The “source_knowledge” variable stores all the necessary knowledge from the Pinecone database to be used in the prompt.

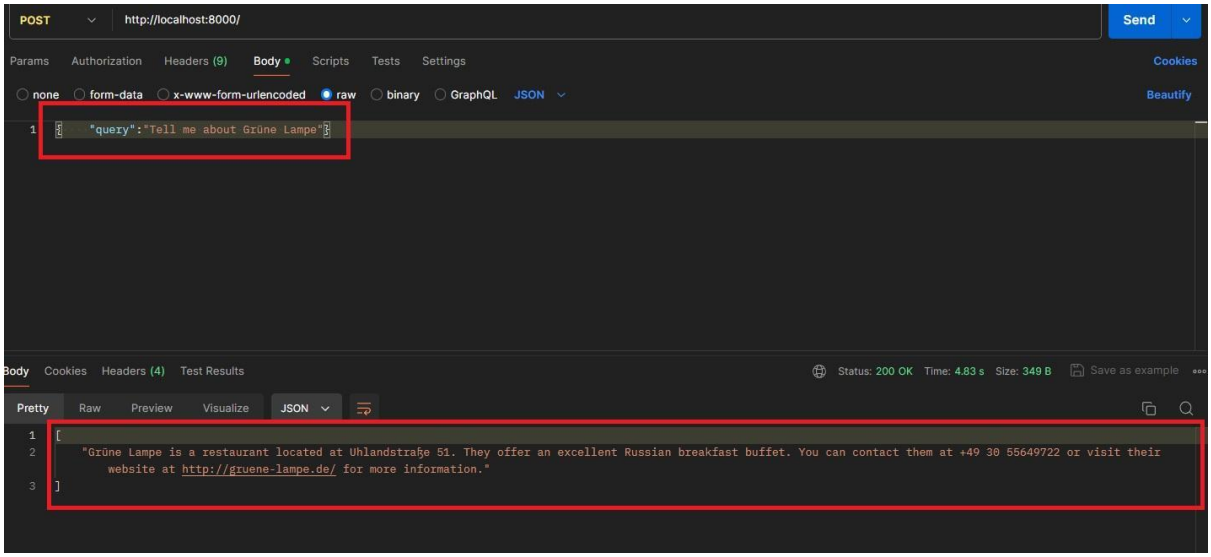
Defining the API endpoints for creating and deleting queries. The POST endpoint processes incoming queries, generates an appropriate prompt, and retrieves the assistant's response. The DELETE endpoint clears the conversation history:

```
@app.post('/')
```

```python
async def create_query(query: Query):
    query_text = query.query
    prompt = HumanMessage(
        content=augmented_prompt(query_text)
    )
    messages.append(prompt)
    res = chat(messages)
    return {res.content}

@app.delete('/')
async def delete():
    global messages
    messages = []
```

Using this API endpoint, the next step is to test the functionality using Postman. Below we can see the API is working as intended and can be used to connect our frontend application.



5.6.Figure 12 API response

Creating and connecting frontend to API

To connect the API to the frontend of TravelTalk, we will integrate a new component called "ChatComponent.vue" into the existing VueJS application. Modern frontend libraries like ReactJS and VueJS require the first character of the file name to be capitalized. This convention ensures that the JSX (JavaScript XML) rendering agent can identify the component as an HTML element during the import process, instead of mistaking it for a regular variable.

A VueJS file typically consists of three main sections: the template, script, and style. The template section is where the HTML markup resides, defining the structure and layout of the component. The script section contains the methods and JavaScript logic, which control the component's behaviour and data manipulation. Lastly, the style section includes the CSS styling, which governs the visual appearance of the component. Below is an example of how the "ChatComponent.vue" file is structured:

```
<template>
  <div>
    <button @click="toggleChat" class="toggle-button">
      {{ isOpen ? "Close Chat" : "Open Chat" }}
    </button>
    <div v-if="isOpen" class="chat-container">
      <div class="messages">
        <div v-for="(message, index) in messages" :key="index"
class="message">
          <span :class="message.type">{{ message.content }}</span>
        </div>
        <div v-if="loading" class="message bot">
          <span>...</span>
        </div>
      </div>
      <div class="input-container">
        <input
          v-model="newMessage"
          @keyup.enter="sendMessage"
          placeholder="Type your message here..."
        />
        <button @click="sendMessage">Send</button>
      </div>
    </div>
  </div>
</template>
```

In this template, we define the structure of the chat interface. A button is used to toggle the chat window open and closed. When the chat is open, a container displays the messages and an input field for the user to type new messages. The v-if directive is used to conditionally render elements based on the component's data properties, such as “isOpen” and “loading”.

```
<script>
import axios from "axios";

export default {
  data() {
    return {
      messages: [],
      newMessage: "",
      isOpen: false,
      loading: false,
    };
  },
  methods: {
    toggleChat() {
      if (this.isOpen) {
        this.closeChat();
```

```
      } else {
        this.isOpen = !this.isOpen;
      }
    },
    async closeChat() {
      try {
        await axios.delete("http://localhost:8000/");
        this.isOpen = false;
        this.messages = [];
      } catch (error) {
        console.error("Error closing chat:", error);
      }
    },
    async sendMessage() {
      if (this.newMessage.trim() === "") {
        return;
      }

      const userMessage = {
        content: this.newMessage,
        type: "user",
      };

      this.messages.push(userMessage);
      this.newMessage = "";

      this.loading = true;

      try {
        const response = await axios.post("http://localhost:8000/", {
          query: userMessage.content,
        });

        const botMessage = {
          content: response.data[0],
          type: "bot",
        };

        this.messages.push(botMessage);
      } catch (error) {
        console.error("Error sending message:", error);
      } finally {
        this.loading = false;
      }
    },
  },
};
</script>
```

In the script section, we import axios to handle HTTP requests. The component's data function returns an object containing the state properties: “messages” (an array to hold chat messages), “newMessage” (the current input from the user), “isOpen” (a Boolean to control the visibility of the chat window), and “loading” (a Boolean to indicate if a message is being processed).

The “toggleChat” method toggles the chat window's visibility. If the chat is open, it calls the “closeChat” method, which sends a DELETE request to the backend to close the session and clear the messages. The “sendMessage” method is responsible for sending user messages to the backend. It first checks if the input is not empty, then adds the user message to the messages array and resets the “newMessage” input. The method sets loading to true and sends a POST request with the user's message. Upon receiving the response, it adds the bot's reply to the messages array and sets loading to false.

With the component fully built and imported into the “PostComponent.vue” file, we can finally visualise r RAG-extended travel assistant functioning on the TravelTalk application. This integration signifies the completion of the project. The system is now capable of engaging with users, providing dynamic and contextually relevant travel advice, showcasing the practical application of advanced AI techniques in a user-friendly interface.

Figure 13 is the visualisation of RAG-extended travel assistant,

5.7. Figure 13 Frontend implementation of RAG extended chatbotPrompt engineering

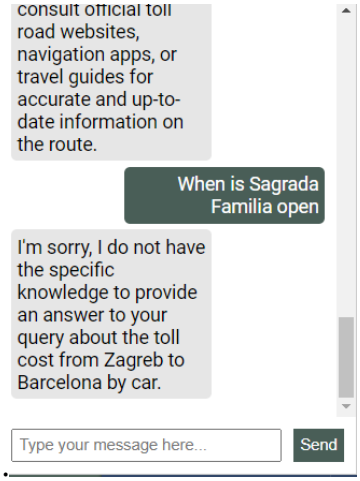
During the process of testing the RAG-extended travel assistant, prompt engineering proved to be a crucial step in achieving desirable results. Throughout the testing phase, numerous instances arose where the retrieval system failed to properly locate the requested data. For example, a query such as “`List some restaurants in Berlin`” would sometimes yield

irrelevant or incorrect responses like "`I don't have any travel-related information for Paris or Rome.`" Clearly, these results did not meet the requirements for a successful travel assistant. Addressing these types of issues was essential for ensuring the reliability and accuracy of the travel assistant. But how can such problems be resolved? The answer lies in matriculate prompt engineering.

Prompt engineering involves the strategic design and refinement of the input prompts given to a language model to elicit the most accurate and relevant responses. This process includes crafting precise, unambiguous prompts and iteratively testing and adjusting them to minimize errors and maximize the quality of the model's output [29]. Effective prompt engineering can significantly enhance the performance of language models, particularly in tasks requiring specific and accurate information retrieval, as seen with the travel assistant. By carefully tuning the prompts, developers can guide the model to better understand the context and deliver responses that align with user expectations, thereby improving the overall utility and user experience of the application.

During the testing of the RAG-extended travel assistant, prompt engineering involved creating and refining system prompts that instruct the language model on how to behave and what information to prioritize. These system prompts are meta-instructions provided to the model to ensure it understands the type of information it should retrieve and how to present it. For instance, the system prompt for this project is crafted to say, "`You are a helpful and friendly travel assistant. Using 10th grade knowledge, you will answer user queries about anything related to traveling. Questions about cities, navigation, cuisine etc. If a question arises about a different area, respectfully decline the query.`"

The biggest boost in the agent's accuracy was further cleaning the data, increasing the chunk size and converting all characters to lower-case. Below in Figures 14 and 15 are examples of hallucinations, and improper retrieval respectively.



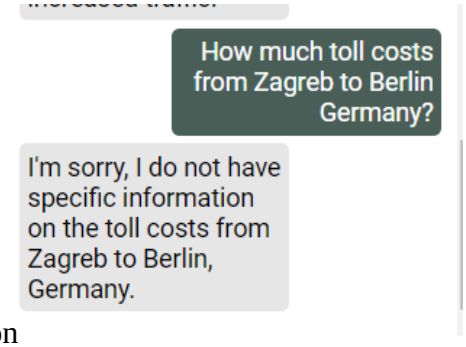


Figure 14 Example of hallucination

Figure 15 Example of improper retrieval

6. Results

The effectiveness of the RAG-extended travel agent was tested and evaluated by inputting the exact same prompt into both the travel agent and ChatGPT-4.0, and then comparing the results. Below in Table 1 are shown all evaluations, highlighting differences in their performance and accuracy. This comparison aims to compare the strengths and weaknesses of both systems (e.g. RAG-extended responses vs. ChatGPT-4o, responses) in handling user queries effectively. The full responses are provided in Figures 16 - 21.

Table 1Evaluation of responses for RAG extended assistant and ChatGPT

Input	**RAG-extended travel assistant response**	**ChatGPT-4o response**
"List some restaurants in Alicante"	Answer satisfies the user query and provides factual, and specific knowledge (address,	ChatGPT provided a broader response that still satisfied the user query but didn't provide any specific or factual information.

	phone numbers, prices etc.) For full response see Figure 16	For full response See Figure 17
"Tell me about Nemesio Casa"	The answer provided was very brief, yet it was very useful. The answer provided contacts and links to external websites. For full response See Figure 18	The answer provided was very detailed, but didn't provide information that the travel assistant provided. The answer wasn't focused on giving specific information. For full response See Figure 19
"Can I take a train to Dubrovnik?"	Brief answer with very helpful and specific information. For full response See Figure 20	Very detailed and helpful answer. Provided the same information the travel assistant provided, but in more detail. For full response See Figure 21

The travel assistant consistently provided briefer but accurate responses, offering specific details such as addresses, phone numbers, and useful links. In contrast, ChatGPT's responses were more detailed but often lacked the concise, factual information that the travel assistant provided. The full responses to each query are in the Figures 16-21.

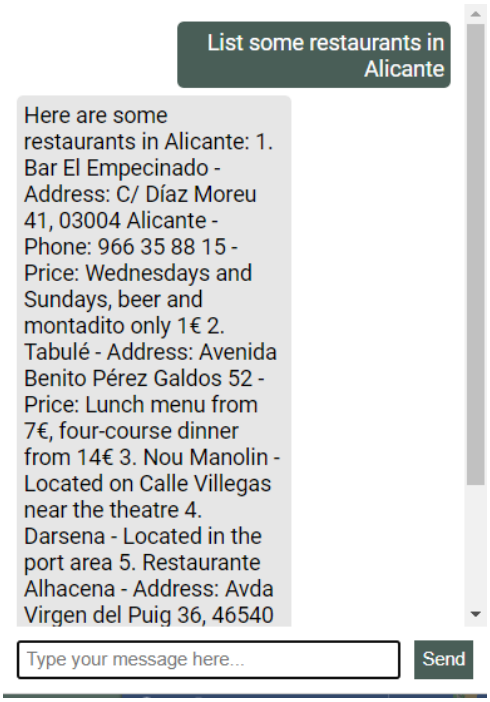


Figure 16 Input 1" List some restaurants in Alicante"" - Travel assistant response

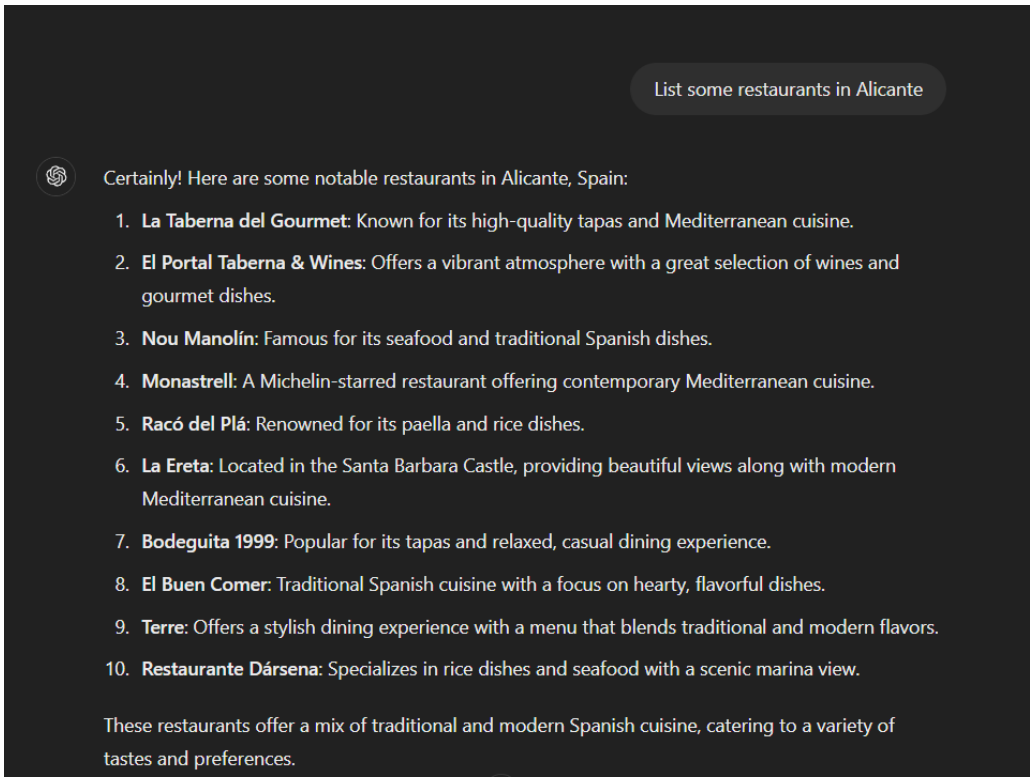


Figure 17 Input 1: "List some restaurants in Alicante" - ChatGPT-4o response

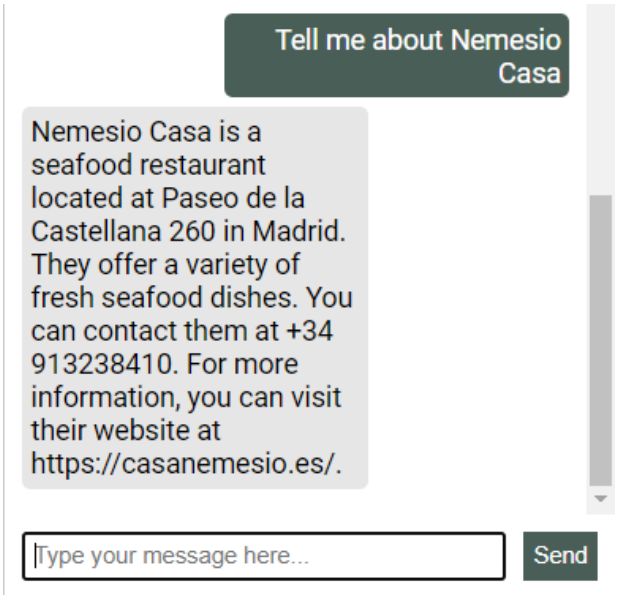


Figure 18 Input 2 "Tell me about Nemesio Casa" - Travel assistant response

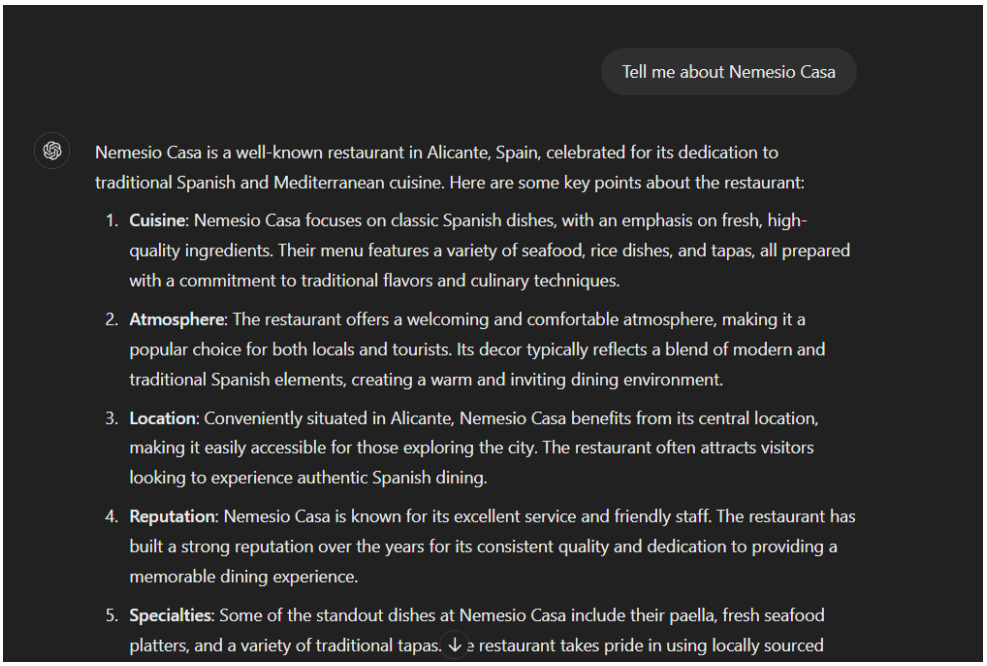


Figure 19 Input 2 "Tell me about Nemesio Casa"" - ChatGPT-4o response

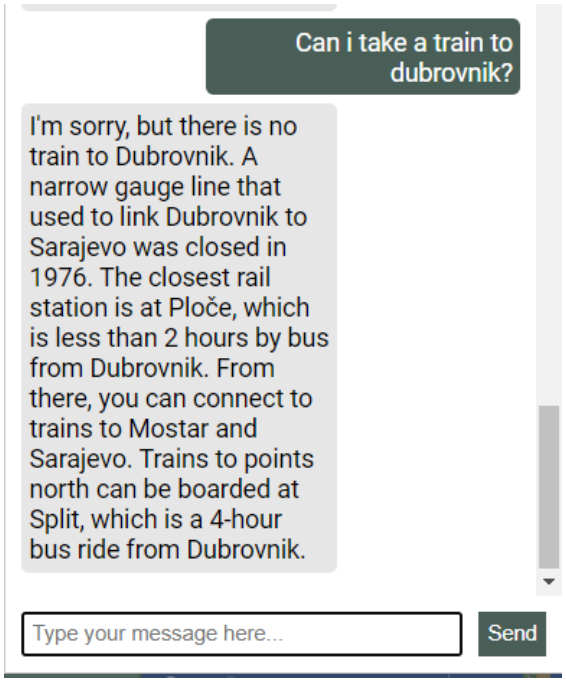


Figure 20 Input 3" "Can I take a train to Dubrovnik?"" - Travel assistant response

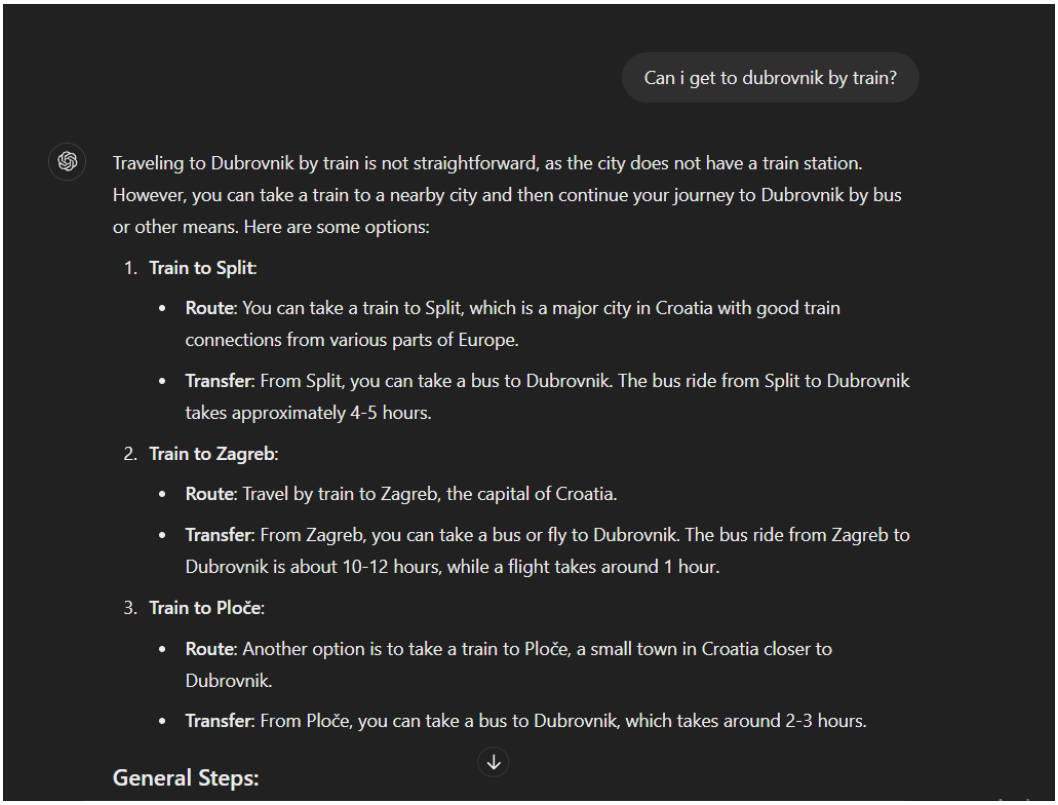


Figure 21 Input 3"Can I take a train to Dubrovnik?" - ChatGPT-4o response

7. Conclusion

In this thesis, the process of implementing Retrieval Augmented Generation (RAG) with a large language model and providing a frontend solution in TravelTalk application was implemented. Key concepts such as hallucinations, vector embeddings, and other relevant mechanisms were thoroughly explained to provide a solid theoretical foundation for RAG implementation. Through testing and evaluations, it was shown that the RAG-extended travel assistant generally outperformed a cutting-edge large language model (ChatGPT-4.0). The results indicate that RAG not only enhanced the accuracy and specificity of responses but also effectively mitigated common issues such as hallucinations and irrelevant information.

In conclusion, the findings of this thesis strongly support the benefit of integration of RAG when building products or applications that utilize large language models. The implementation of RAG offers significant benefits, including improved performance, greater reliability, and more precise information retrieval, which are crucial for user satisfaction. Given these advantages, it is highly recommended to employ RAG in any application leveraging large language models.

Future development plans involve several key areas. First, the database will be expanded to cover global data, allowing for precise and comprehensive data retrieval across different regions. Second, the API will be optimized to enhance performance, focusing on reducing response times and lowering memory usage to ensure efficient operation. Finally, the application will be fully deployed for public use, providing access to all users. These improvements are intended to support scalability, performance, and accessibility.

8. Literature

[1] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS'20)*, 2020, pp. 9459-9474. doi: 10.48550/arXiv.2005.11401.

[2] P. Ji et al., "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions," 2023. [Online]. Available: https://arxiv.org/abs/2304.07857.

[3] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal LSH for angular distance," in *Advances in Neural Information Processing Systems*, vol. 30, pp. 1225-1233, 2018.

[7] A. Vaswani et al., "Attention Is All You Need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*, Long Beach, CA, USA, 2017, pp. 6000-6010. doi: https://dl.acm.org/doi/10.5555/3295222.3295349

[8] T. Brown et al., "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877-1901, 2020. doi: 10.48550/arXiv.2005.14165.

[10] A. Radford et al., "Improving Language Understanding by Generative Pre-Training," OpenAI, 2018. [Online]. Available: https://www.semanticscholar.org/paper/Improving-Language-Understanding-by-Generative-Radford-Narasimhan/4856ef15e8bb2b3a85bbaab0e9ac1540e9b981a2.

[11] Alabdulmohsin I., Lucic M., “A Near-Optimal Algorithm for Debiasing Trained Machine Learning Models” 2021. [Online], Available: https://arxiv.org/abs/2106.12887

[12] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In Proceedings of the 27th Annual Conference on Neural Information Processing Systems, pages 3111– 3119, 2013.

[13] Malkov, Y. A., & Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *42*(4), 824-836.

[14] A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge

[15] Understanding Langchains RecursiveCharacterTextSplitter : https://dev.to/eteimz/understanding-langchains-recursivecharactertextsplitter-2846

[16] Web framework benchmark: https://www.techempower.com/benchmarks/#hw=ph&test=fortune&section=data-r22

[17] Python Starlette framework “Official web page for the Starlette framework”. https://www.starlette.io/ (accessed August 23, 2024.)
[18] Pydantic framework “Official documentation page for the pydantic framework”. https://docs.pydantic.dev/latest/ (accessed August 23, 2024.)
[19] Large Language Models: A Survey Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu Richard Socher, Xavier Amatriain, Jianfeng Gao
[20] Vuejs JavaScript framework “Official introductory web page for VueJs”. https://vuejs.org/guide/introduction.html (accessed June 27, 2024)
[31] MongoDB atlas : https://www.mongodb.com/products/platform/atlas-database
[21] NodeJs Official website “Official NodeJs documentation detailing the V8 engine”. https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine (accessed August 19, 2024)
[22] Wikitravel website “Wikitravel main page”. https://wikitravel.org/en/Main_Page (accessed June 19, 2024)
[23] LangcChain website “Main page for LangChain framework”. https://www.langchain.com/ (accessed June 20, 2024)
[24] Langchain paper : An Effective Query System Using LLMs and LangChain https://www.researchgate.net/publication/372529063_An_Effective_Query_System_Using_LLMs_and_LangChain
[25] Pinecone website “Official main page for PineCone vector database” . https://www.pinecone.io/ (accessed August 16, 2024)
[26] FastAPI, "FastAPI dependancy documentation," https://fastapi.tiangolo.com/tutorial/dependencies/ (accessed Aug. 23, 2024).
[27] FastAPI, “Fast API security documentation article” https://fastapi.tiangolo.com/tutorial/security/ (accessed Aug 23, 2024)
[28] Chunking mechanisms and learning , Fernard Gobet (2012) https://www.researchgate.net/publication/308158087_Chunking_mechanisms_and_learning
[29] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications," arXiv preprint arXiv:2402.01234, Feb. 2024. [Online]. Available: https://arxiv.org/abs/2402.01234
[30] Pythorch “Pythorch official web page” https://pytorch.org/ (accessed Aug, 23, 2024)
[31] Acler, M. “The 50 Most Visited Cities in Europe — A Data-Driven Guide for Travel Bloggers” https://medium.com/online-travel-partners/the-50-most-visited-cities-in-europe-a-data-driven-guide-for-travel-bloggers-5ebeaa6e24cb (accessed June 16, 2024)
[32] Z. Wang *et al.*, "Speculative RAG: Enhancing Retrieval Augmented Generation through Drafting," presented at the *[Arxiv, 2024]*.
[33] Zendesk, “Zendesk official AI trust page with full privacy disclosure”, https://www.zendesk.com/trust-center/#ai (accessed, Aug 21, 2024)
[34] IBM Watsonx Assistant, “Official web page detailing IBM’s conversational AI ”, https://www.ibm.com/products/watsonx-assistant (accessed Aug 21, 2024)
[35] CopyAI AI chat, “AI Chat: Your GTM team can do more with less
”, (accessed July 16, 2024)

[36] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," arXiv preprint arXiv:1810.04805, Oct. 2018, revised May 2019.
[37] Pinecone security page, “Trust and security”, https://www.pinecone.io/security/ , (accessed Aug 21, 2024)
[38] Efimov V., “Similarity Search, Part 4: Hierarchical Navigable Small World (HNSW)”, https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37, (accessed Aug 23, 2024)

9. List of tables

10. List of illustrations

11. List of attachments

Attachment 1: TravelTalk GitHub repository, https://github.com/Sventajz/Travel.talk-WIP-

Attachment 2: RAG extended travel assistant GitHub repository, https://github.com/Sventajz/rag_api