

Transformacija poslovnih sustava pomoću velikih jezičnih modela i metodi RAG

Horvat, Dora

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: <https://urn.nsk.hr/urn:nbn:hr:168:830289>

Rights / Prava: [In copyright](#)/[Zaštićeno autorskim pravom.](#)

Download date / Datum preuzimanja: **2025-03-21**



Repository / Repozitorij:

[FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repository](#)



UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 637

**TRANSFORMATION OF BUSINESS SYSTEMS USING LARGE
LANGUAGE MODELS AND THE RAG METHOD**

Dora Horvat

Zagreb, June 2024

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 637

**TRANSFORMATION OF BUSINESS SYSTEMS USING LARGE
LANGUAGE MODELS AND THE RAG METHOD**

Dora Horvat

Zagreb, June 2024

MASTER THESIS ASSIGNMENT No. 637

Student: **Dora Horvat (0036517299)**

Study: Computing

Profile: Data Science

Mentor: prof. Ivica Botički

Title: **Transformation of Business Systems using Large Language Models and the RAG Method**

Description:

As part of the thesis, a system prototype will be developed that enables better user search of information within business systems by integrating a platform which is used as a business knowledge base with a large language model extended by the RAG (Retrieval Augmented Generation) method. As part of the thesis, the RAG method which enables the integration of external sources of information at the time of use of large language models resulting in faster and more efficient access to internal business information will be described. Embedding models will be analysed and evaluated, with the selection of the most suitable model tailored to the performance and specifics of the business environment. Vector databases, semantic search methods, and retrieval of related documents will be explored, leading to the choice of a suitable vector database for implementing the RAG method. The pre-trained models that are currently available will be reviewed, and parameters and computational resources will be evaluated before the model implementation strategy is chosen. Test instances will be proposed and designed, and the developed prototype will be analysed, and the results presented.

Submission date: 28 June 2024

DIPLOMSKI ZADATAK br. 637

Pristupnica: **Dora Horvat (0036517299)**

Studij: Računarstvo

Profil: Znanost o podacima

Mentor: prof. dr. sc. Ivica Botički

Zadatak: **Transformacija poslovnih sustava pomoću velikih jezičnih modela i metodi RAG**

Opis zadatka:

U sklopu diplomskog rada razvit će se prototip sustava koji omogućuje bolje korisničko pretraživanje informacija unutar poslovnog sustava integrirajući platformu koja se koristi kao baza poslovnog znanja s velikim jezičnim modelom proširenim RAG (Retrieval Augmented Generation) metodom. U sklopu rada opisat će se metoda RAG koja omogućuje integraciju vanjskih izvora informacija u trenutku korištenja velikih jezičnih modela, što rezultira bržim i učinkovitijim pristupom internim poslovnim informacijama. Razradit će se i modeli ugradnje, prilagođeni performansama i specifičnostima poslovnog okruženja. Istražit će se vektorske baze, način semantičkog pretraživanja i dohvaćanja povezanih dokumenata iz poslovnih sustava te će se odabrati prikladna vektorska baza za implementaciju metode RAG. Napravit će se pregled dostupnih prethodno naučenih modela, procjena broja parametara i potrebnih računalnih resursa te će se odabrati način implementacije modela. Predložiti će se i oblikovati ispitni primjerci te će se provesti analiza razvijenog prototipa i predstaviti ostvareni rezultati.

Rok za predaju rada: 28. lipnja 2024.

Iskreno se zahvaljujem svojem mentoru prof. dr. sc. Ivici Botičkom na njegovoj podršci i poticaju pri izradi ovog rada.

Svojoj obitelji i dečku, a posebno mojem djedu, čijim sam stopama koračala, na ljubavi i podršci tijekom cijelog studija.

Svojim kolegama iz Syntia, bez kojih ovaj rad ne bi bio moguć, zahvaljujem se na neizmjernej pomoći, savjetima i suradnji.

Konačno, svojim prijateljima, koji su učinili ovo akademsko putovanje nezaboravnim i ispunjenim.

Contents

1	Introduction	3
2	Theoretical Overview	5
2.1	The Retrieval Augmented Generation (RAG) Method	5
2.2	Vector Embeddings	7
2.3	Vector Databases	8
2.4	Large Language Models	12
3	System Design	15
3.1	Architecture	15
3.2	System Requirements	16
3.2.1	Vector Database	16
3.2.2	Large Language Model (LLM)	17
3.2.3	Embedding Model	18
3.3	Components	19
3.3.1	Qdrant	19
3.3.2	Mistral 7B	21
3.3.3	Sentence-transformers model: all-mpnet-base-v2	22
4	Implementation	24
4.1	Indexing Phase	25
4.1.1	Document Loading	25
4.1.2	Text Chunking	28
4.1.3	Embedding and Storing	31
4.2	Generative Phase	31
4.2.1	Retrieval	32

4.2.2	Generation	35
4.3	User Interface	37
4.4	Deployment	38
4.4.1	Google Cloud Platform	38
4.4.2	Deploying the System	39
5	Results	40
5.1	RAG Evaluation	40
5.1.1	System Testing	41
6	Discussion	46
6.1	Implications	46
6.2	Future Direction	46
6.3	Future of RAG	48
7	Conclusion	49
	References	50
	Abstract	54
	Sažetak	55

1 Introduction

Recently, the widespread adoption of Large Language Models (LLMs) has revolutionized natural language processing, primarily with the emergence of large language models such as GPT models, PaLM, Llama, and others. For instance, ChatGPT, a tool based on GPT models, has managed to attract over a hundred million monthly users in less than a year [1]. By leveraging a massive number of parameters, ranging from tens of millions to several billion, these models have demonstrated exceptional capabilities in solving a wide range of natural language processing tasks. Despite their remarkable abilities, LLMs come with certain limitations. One inherent issue is their reliance on the knowledge they are trained on; once the model training is completed, the set of data the model can use to generate responses to queries becomes fixed. Without continuous retraining on "fresh" data, such models can quickly become outdated. Additionally, they are often trained on a broad and publicly available set of "general" data, which may be inadequate for queries from a specific domain. In situations where there is a gap in their knowledge for any reason, LLMs extrapolate, producing incorrect but convincing statements—a phenomenon now known as hallucination. In business domains, the limitations of language models are particularly pronounced due to the nature of the data they must use. Businesses and technology companies store critical information within internal documents, which constitute the foundational knowledge of the organization. Unlike publicly available information, such internal data is excluded from the training process of public LLMs, leading to a significant performance gap when it comes to queries specific to a particular domain.

As one solution to these limitations, model fine-tuning can be applied. Model fine-tuning involves using a previously trained language model and further training it on a smaller dataset specific to a particular domain. This allows the model to adapt to specific business needs, reducing challenges arising from outdated or missing data. How-

ever, this method is often time and resource-intensive, making the entire process costly. Moreover, continuously fine-tuning the model with the latest data is impractical (and often infeasible), especially for models with a substantial number of parameters working with a large set of variable data. Consequently, there is a growing interest in a different approach, based on information retrieval—Retrieval Augmented Generation (RAG). First proposed by Lewis et al. in [2], this approach involves using external data stores in real-time— at the moment of generating a response to a query—to finally obtain an answer enriched with a combination of context and recent knowledge. Based on the user’s query, relevant contextual information is retrieved, which is then merged with the user input to create a richer query containing contextual information not otherwise available to the LLM.

In business environments, where accessing internal information is often challenging, especially when working with a knowledge database of substantial size, traditional keyword-based search methods often fall short when it comes to retrieving specific and complex information crucial for business processes. The keyword search can result in imprecise or overly generalized results, slowing down access to necessary information. Managing a large database of internal information requires more than a simple keyword search, and advanced retrieval methods are often needed to identify and extract relevant information from the abundance of data quickly. By combining the RAG method with the use of LLMs, the integration of external sources of knowledge at the point of inference enables more precise, faster, and more efficient access to information. Although the use of the RAG method to enrich LLM models is still in development, available research suggests that retrieval-based models can outperform traditional parameter-based models without retrieval by using fewer parameters [3], can update their knowledge by swapping their corpora for retrieval [4], and can provide citations to users to easily verify and assess predictions [5, 6].

2 Theoretical Overview

2.1 The Retrieval Augmented Generation (RAG) Method

The method of Retrieval-Augmented Generation (RAG), originally proposed by Facebook AI Research (FAIR) in [2], introduced a new way to enhance pre-trained language models, especially for tasks that are specific to particular fields and domains. Although RAG is a relatively new approach in the field of Natural Language Processing (NLP), its impact is already evident through its integration into services like Bing Search [7], highlighting not only its theoretical but also its practical capabilities.

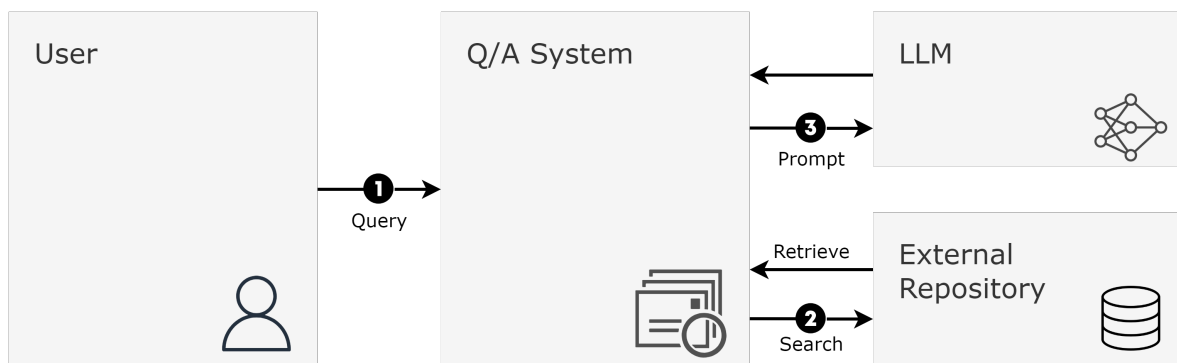


Figure 2.1: RAG flow

The RAG method is based on the principle of *in-context learning*: the ability of a large language model to leverage context within the prompt to create a better output [8]. In contrast to classic LLM querying, where the input query is ingested directly by the language model, RAG combines LLM with a non-parametric data source that contains specific and up-to-date information. On a high level, this process involves several steps, presented in 2.1 The user initially inputs a query into the question-answer (Q/A) system, which then searches for relevant documents by sending the query to an external data repository that serves as a contextual knowledge base that the LLM has not been

previously trained on. The retrieved contextual information is then combined with the initial user query to construct a new, enriched query for the large language model.

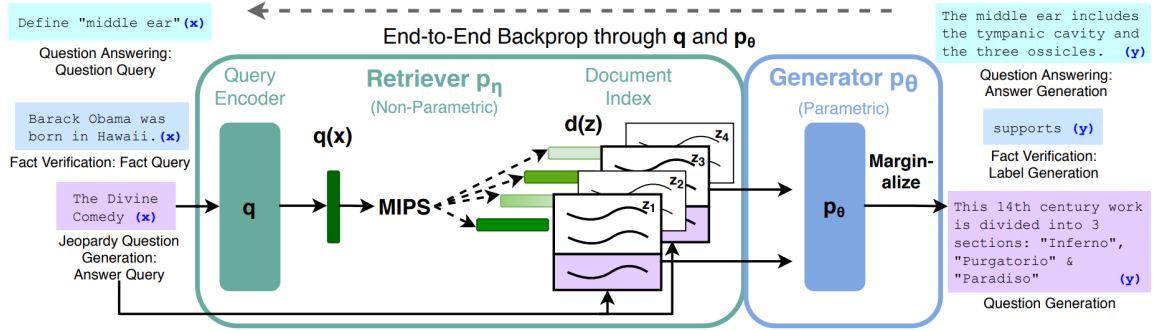


Figure 1: Overview of our approach. We combine a pre-trained retriever (*Query Encoder + Document Index*) with a pre-trained seq2seq model (*Generator*) and fine-tune end-to-end. For query x , we use Maximum Inner Product Search (MIPS) to find the top-K documents z_i . For final prediction y , we treat z as a latent variable and marginalize over seq2seq predictions given different documents.

Figure 2.2: RAG architecture presented in [2]

In [2], the proposed RAG model consists of a parametric memory - a pre-trained seq2seq transformer, and the non-parametric memory - a dense vector index accessed with a pre-trained neural retriever. The RAG architecture, as depicted in 2.2, is built out of two main components: the retriever and the generator. The retriever component plays a crucial role in transforming the user's text into vector forms, as well as the text of contextual documents, effectively creating a searchable index. The retrieval process includes vectorizing the input text, searching for related and similar documents in the index, and converting the vectors of retrieved documents back into textual form. The second component, the generator, takes the user's input query and the retrieved document text to generate a new "prompt" for the large language model. Formally, RAG considers an input sequence \mathbf{x} (the prompt) and uses this input to retrieve documents \mathbf{z} (text chunks), which are used as context when generating a target sequence \mathbf{y} . For retrieval, authors in [2] use the dense passage retrieval (DPR) model, a pre-trained bi-encoder that uses separate BERT models to encode queries (query encoder) and documents (document encoder). DPR model follows a bi-encoder architecture:

$$p_{\eta}(z|x) \propto \exp(\mathbf{d}(z)^{\top} \mathbf{q}(x))$$

$$\mathbf{d}(z) = \text{BERT}_d(z)$$

$$\mathbf{q}(x) = \text{BERT}_q(x)$$

where $\mathbf{d}(z)$ is a dense representation of a document produced by a $\text{BERT}_{\text{BASE}}$ *document encoder*, and $\mathbf{q}(x)$, a query representation produced by a *query encoder*, also based on $\text{BERT}_{\text{BASE}}$. The retrieved document probability is proportional to the inner product of query and document embeddings. The generator component $p_{\theta}(y_i | x, z, y_{1:i-1})$ is modelled by a pre-trained BART model. Parametrized by θ , the generator component generates a current token based on a context of the previous $i - 1$ tokens $y_{1:i-1}$, the original input x and a retrieved passage z .

In practice, implementations of the RAG method leverage advancements in vector databases and LLMs such as GPT models. The retriever component is most often a vector database, which can store vector embeddings and perform similarity search. Advanced LLMs such as GPT models are used as the generative component. These models are pre-trained on a large amount of data, providing superior generation capabilities.

2.2 Vector Embeddings

Vector embeddings are central to the field of natural language processing (NLP) and the development of large language models. They represent words, sentences, or documents as high-dimensional vectors within a continuous vector space. Each dimension in the vector space captures different aspects of the meaning or context of the text. These vector representations make it possible to translate semantic similarity to proximity in a vector space [9]. This is crucial for RAG systems; given a query, the system can quickly find and retrieve passages with similar semantic content by comparing the vector representations. Embedding methods for textual data have greatly improved over the last decade, from

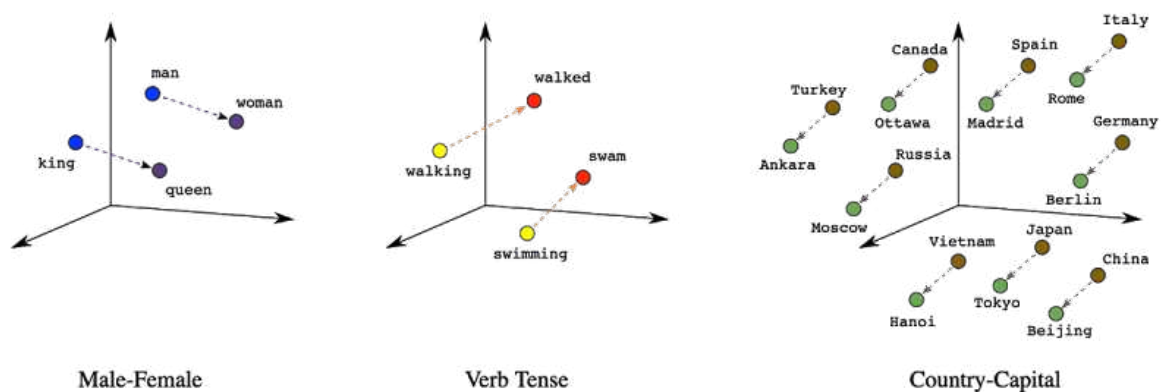


Figure 2.3: Simplified visualization of semantic similarity of vectors generated by word2vec [10]

Google’s word2vec to transformer models. Word2vec model introduced the idea of *dense* vectors, in which values are non-zero. Initially trained by Google on 100 billion words, it used a neural network model to learn word associations from a large text corpus. Vector representations created by the model encapsulate semantic similarities between words, so words appearing in similar contexts have vectors that are close in the vector space, as depicted in 2.3. Word2vec focuses on word-level representations and cannot independently generate vectors for longer texts such as sentences, paragraphs, or documents. To represent longer texts, the vectors of individual words can be aggregated, often by applying weightings to emphasize certain words over others. Current state-of-the-art embedding models are based on *transformer* architecture introduced in [11]. Transformer models like BERT and its successors enhance search accuracy, precision, and recall by considering the context of each word to generate fully contextual embeddings. Unlike word2vec embeddings, which do not account for context, transformer-generated embeddings incorporate the entire input text. This means that each instance of a word has a unique embedding influenced by its surrounding text. These contextual embeddings more accurately represent the multiple meanings of words, which can only be clarified when viewed in their specific context.

2.3 Vector Databases

The rise of large language models has directly influenced the rise of vector databases. This can best be observed through the star history on Github for several vector database solutions; the graph in 2.4 shows significant growth following the initial release of ChatGPT in November 2022. The spike in interest emphasizes the increasing reliance on vector databases for handling the high-dimensional data required by LLMs. Vector databases are a specialized category of database systems designed to handle high-dimensional feature vectors generated by embedding models. One of the main differences between vector databases and traditional Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) databases lies in the nature of the data they handle. OLTP databases are designed for managing transactional data; they store the data in a row-based format where each row corresponds to a single record, and each column within the row represents an attribute of the record. This row-oriented storage architecture enables fast access and per-record manipulation, making OLTP databases best suited for a high

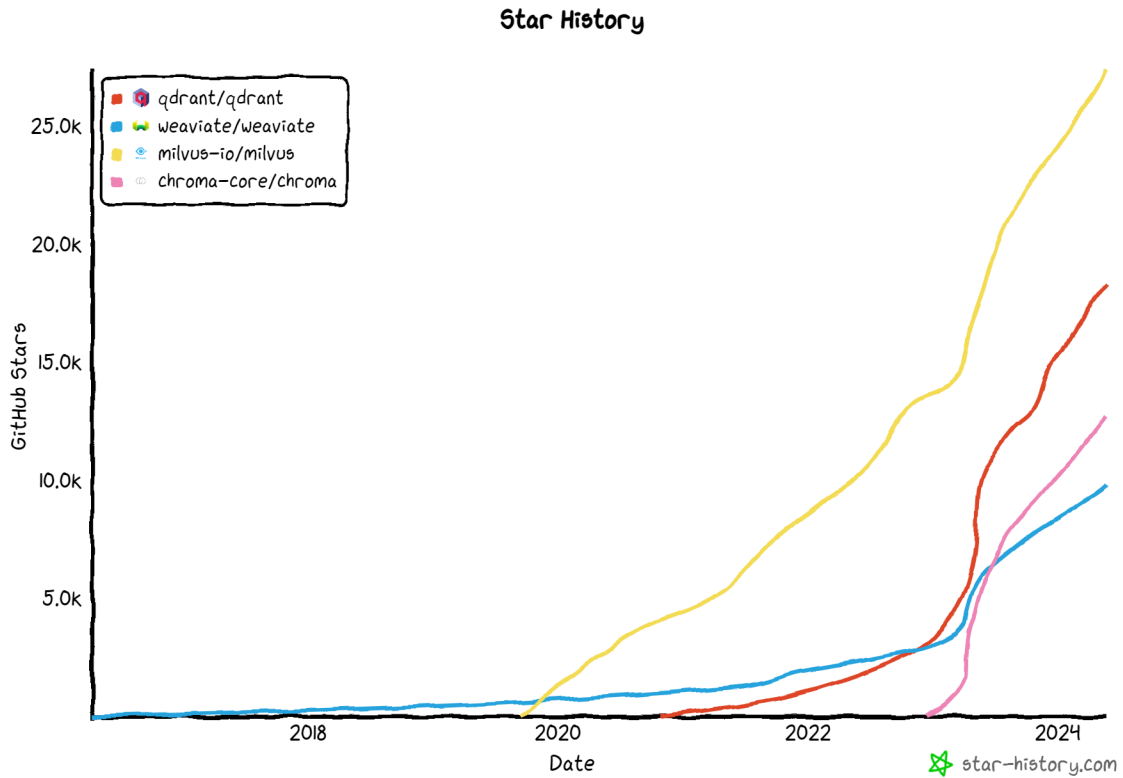


Figure 2.4: GitHub star history for popular vector databases

volume of short, atomic transactions (insertions, updates, and deletions). On the other hand, OLAP databases are designed for analytical processing and data warehousing applications; they store the data in columnar format, where each column holds data for a specific attribute across many rows. The column-oriented storage structure primarily enhances read performance, enabling large-scale analytical queries with complex aggregations and joins. Both OLTP and OLAP systems are not inherently designed to manage high-dimensional vector data. In vector databases, each vector generally corresponds to an item or an entity and is associated with an identifier and a payload containing additional information. Vector queries differ from traditional relational queries, as they are based on the concept of similarity. Computing similarity between high-dimensional vectors is computationally expensive and involves different, specialized techniques that are generally not supported by conventional database systems. These techniques often include approximate nearest neighbor (ANN) search algorithms and specialized indexing methods like Hierarchical Navigable Small World (HNSW) graphs, which are specifically optimized for similarity search.

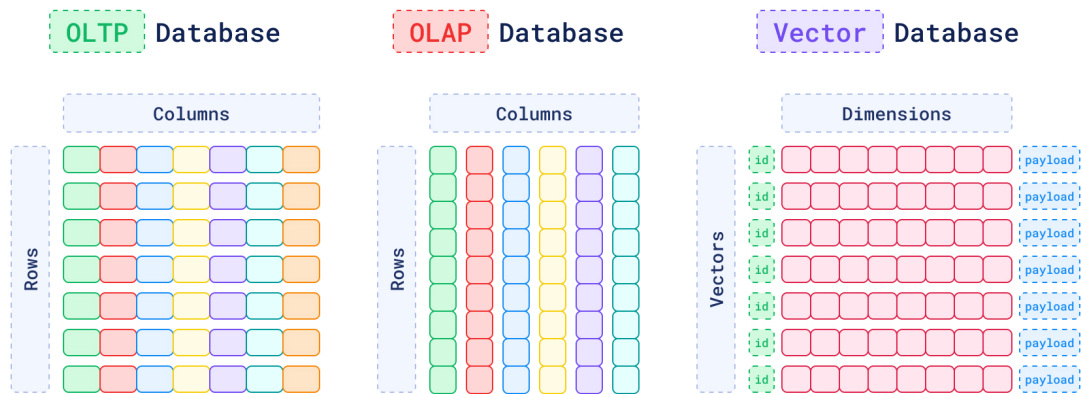


Figure 2.5: Comparison between OLTP, OLAP, and Vector Databases [12]

Query Processing

As mentioned, in vector databases, query processing revolves around the concept of similarity search. The core idea is to identify vectors in a database that are most similar to a given query vector, using various distance metrics to measure this similarity. The most common distance metrics employed in similarity search include Euclidean distance, cosine similarity, and dot product. Euclidean distance measures the straight-line distance between two vectors in the vector space. It is calculated by taking the square root of the sum of the squared differences between corresponding elements of the two vectors. This metric is intuitive and widely used because it directly correlates with our geometric understanding of distance; vectors that are closer together in space have smaller Euclidean distances, indicating higher similarity. Cosine similarity, on the other hand, measures the cosine of the angle between two vectors. This metric evaluates how aligned the vectors are, regardless of their magnitude. It is calculated as the dot product of the vectors divided by the product of their magnitudes. Cosine similarity ranges from -1 to 1, where 1 indicates that the vectors are perfectly aligned (pointing in the same direction), 0 indicates orthogonality (no similarity), and -1 indicates that the vectors are diametrically opposed. This metric is particularly useful in high-dimensional spaces where the magnitude of the vectors might not be as important as their orientation. Lastly, the dot product measures the similarity by directly computing the sum of the products of corresponding elements of the vectors. This metric is effective in scenarios where the magnitude and direction of the vectors both contribute to their similarity. The larger the dot product,

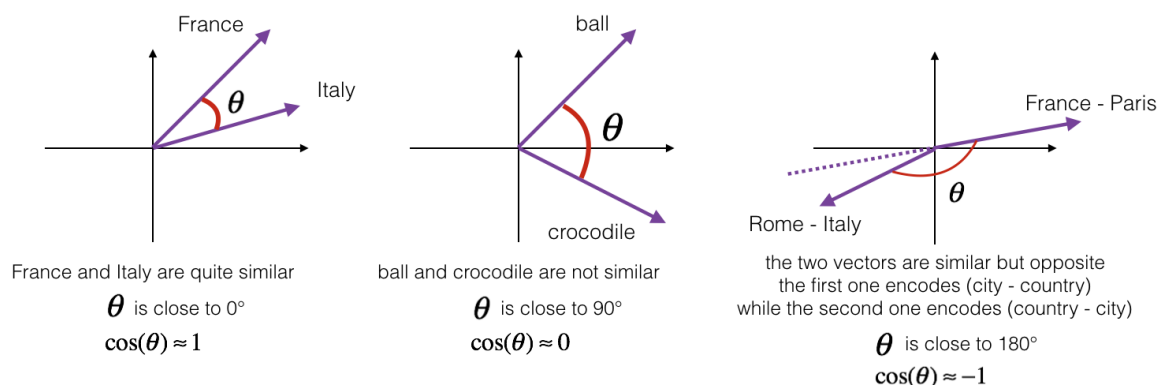


Figure 2.6: Cosine Similarity

the more similar the vectors are. Unlike cosine similarity, the dot product is not normalized, meaning that it can grow with the magnitude of the vectors, which can be useful in certain applications where larger magnitudes imply greater significance.

Indexing

A vector index is a data structure for storing and retrieving high-dimensional vector data, enabling fast similarity searches and nearest-neighbor queries. In traditional databases, we generally query the index for exact matches; but, since vector embeddings capture the semantic meaning of data, the vector index can be searched for approximate matches. The class of algorithms that are used to create and search vector indexing is called Approximate Nearest Neighbor (ANN) search.

The simplest and most straightforward indexing method is flat indexing, where the vectors are simply stored without modifications. The biggest downside of this approach is its speed; since the similarity between the query vectors and every other vector in the index is calculated, the process of retrieval is quite slow. Instead of this “brute force” approach, Locality Sensitive Hashing (LSH) indexes can be constructed. This strategy builds an index using a hashing function; vector embeddings that are near each other (defined by some similarity metric) are hashed to the same bucket. When a query vector is provided, its hash code is matched to a hash code of one of the buckets. A similarity search is then performed on all of the vectors within that bucket. This results in a smaller search space, increasing the speed. Inverted file (IVF) indexes are based on a similar idea as LSH indexes; the query vector is mapped to a smaller subset of the vector space and

a similarity search is performed only on that subset. The difference lies in the method used to create the smaller vector space; in IVF, the original vector space is first clustered, with a centroid for each cluster. For a query vector, the closest centroid is found, and vectors from that cluster are searched. Currently, one of the best ANN algorithms is the Hierarchical Navigable Small World (HNSW) index. It is constructed as a multi-layered graph. At the lowest level, every vector in the vector space is captured. Moving up the layers, points are grouped together based on a similarity metric. Data points in each layer are also connected to data points in the next layer. The highest layer of the graph is the starting point of the search; the closest match from the graph on this layer is taken to the next layer, and this process is continued all the way to the lowest layer.

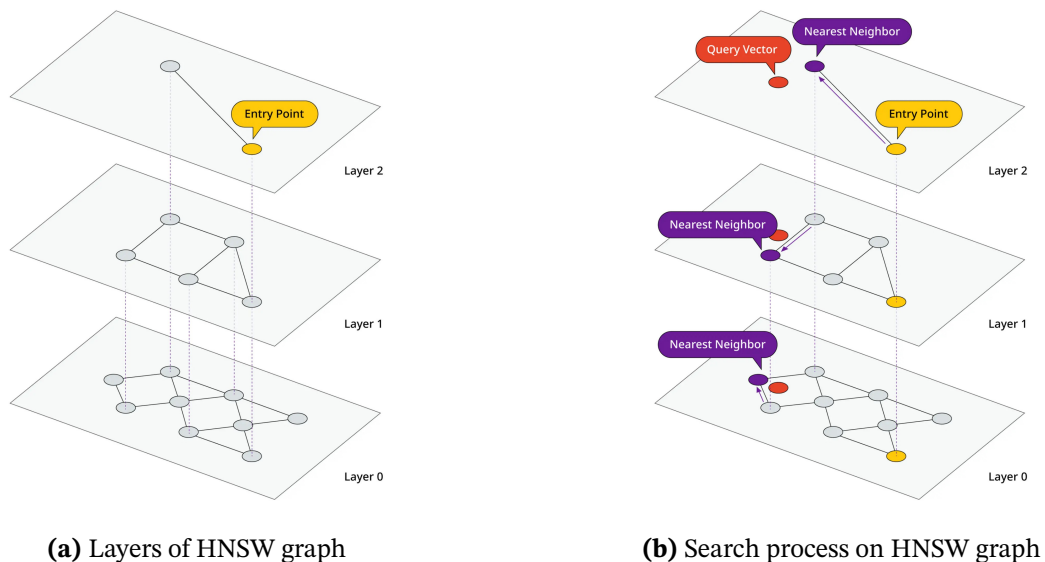


Figure 2.7: Architecture of HNSW index

2.4 Large Language Models

Simply put, large language models are sophisticated AI systems designed to model and process human language at an advanced level. The “large” attribute refers to the number of parameters that define these models; often ranging from hundreds of millions to billions. Their development has been driven by the transformer architecture, a neural network design introduced by Google in the paper “Attention Is All You Need” [11]. Even though transformers are based on encoder-decoder architecture, the transformer architecture fundamentally differs from traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs) in its approach to embedding and processing se-

quences of data. Traditional RNNs and CNNs are unidirectional; they make predictions based only on previous words in a sequence. This limitation can restrict their ability to capture the full context of a sentence. Transformers, on the other hand, utilize an attention mechanism that allows them to consider both previous and following words, making them bidirectional. This bidirectional capability is crucial for understanding context and generating more accurate language representations. Therefore, the transformer architecture consists of an encoder-decoder structure, but with a key innovation: the embedding process is parallelized. In traditional RNNs and CNNs, sequential processing can be slow and computationally intensive because each word must be processed one after the other. Transformers avoid this problem by processing all words in the input sequence simultaneously, greatly enhancing efficiency. The self-attention mechanism enables the model to weigh the importance of different words in a sentence relative to each other. This is achieved through the calculation of attention scores, which determine how much focus each word should receive when generating embeddings. The self-attention process captures the relationships between words, allowing the model to understand context more effectively than traditional models. Research has shown that the initial layers of a transformer focus on understanding the syntax of sentences, such as grammatical structure and word order, while the deeper layers develop a more abstract understanding of the input, capturing semantic meaning and context. The attention mechanism works by first generating three vectors for each word in the input sequence: the query vector, the key vector, and the value vector. These vectors are used to compute the attention scores. The query vector represents the word for which we are seeking context, the key vector represents the words that might contain relevant context, and the value vector represents the actual contextual information. The attention score is calculated as the dot product of the query and key vectors, normalized using a softmax function to ensure the scores sum to one. The resulting attention scores are then used to weigh the value vectors, producing a weighted sum that represents the context-aware embedding of the word. This process is performed in multiple "heads," known as multi-head attention, allowing the model to capture different types of relationships between words simultaneously. The outputs from these heads are then concatenated and passed through a feed-forward neural network, which transforms the attention-weighted embeddings into a format expected by the next layer of the transformer. In the decoder, used during the training phase, there

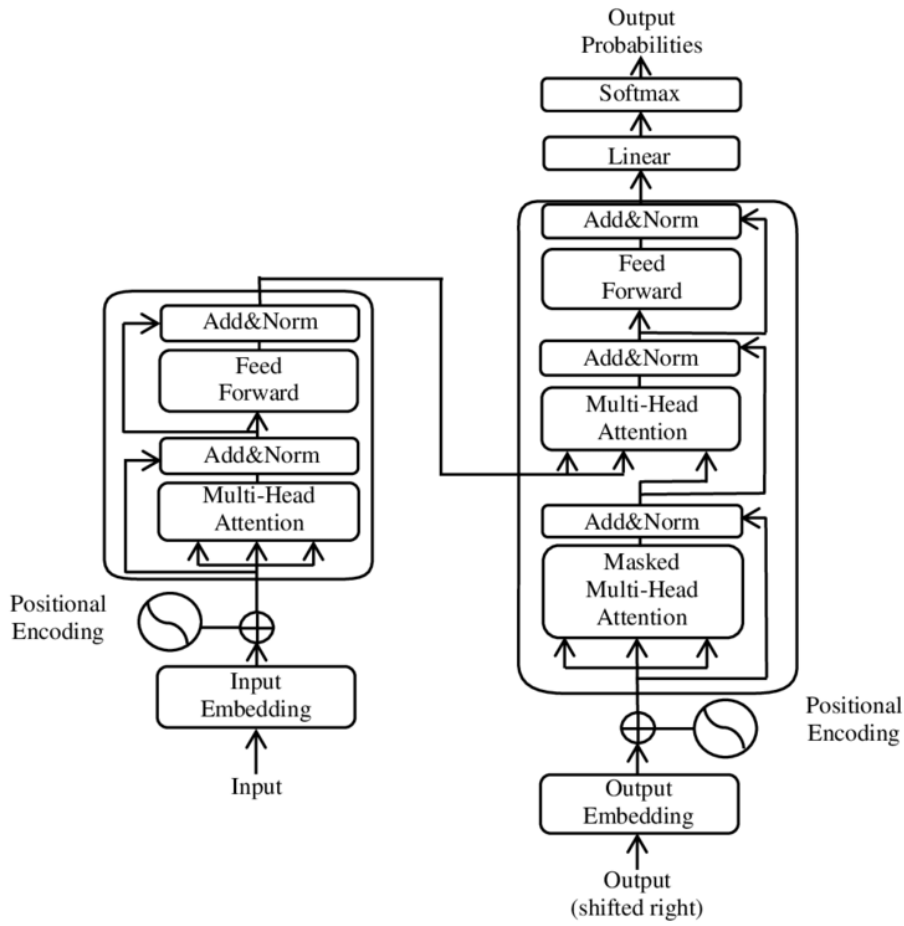


Figure 2.8: Architecture of the Transformer Model [11]

are additional layers of masked multi-head attention, multi-head attention, and position-wise feed-forward networks. The decoder receives two main inputs: the attention vectors from the encoder and the target sentence to be translated. The masked multi-head attention ensures that the decoder can only attend to previous positions, preventing information leakage from future words in the target sentence.

3 System Design

3.1 Architecture

The proposed architecture of the RAG system, depicted in 3.1, builds upon the foundational principles of the RAG method explained in 2.1. The system integrates multiple components, creating a user-oriented framework for leveraging external knowledge sources and a large language model to enhance business information retrieval. At the

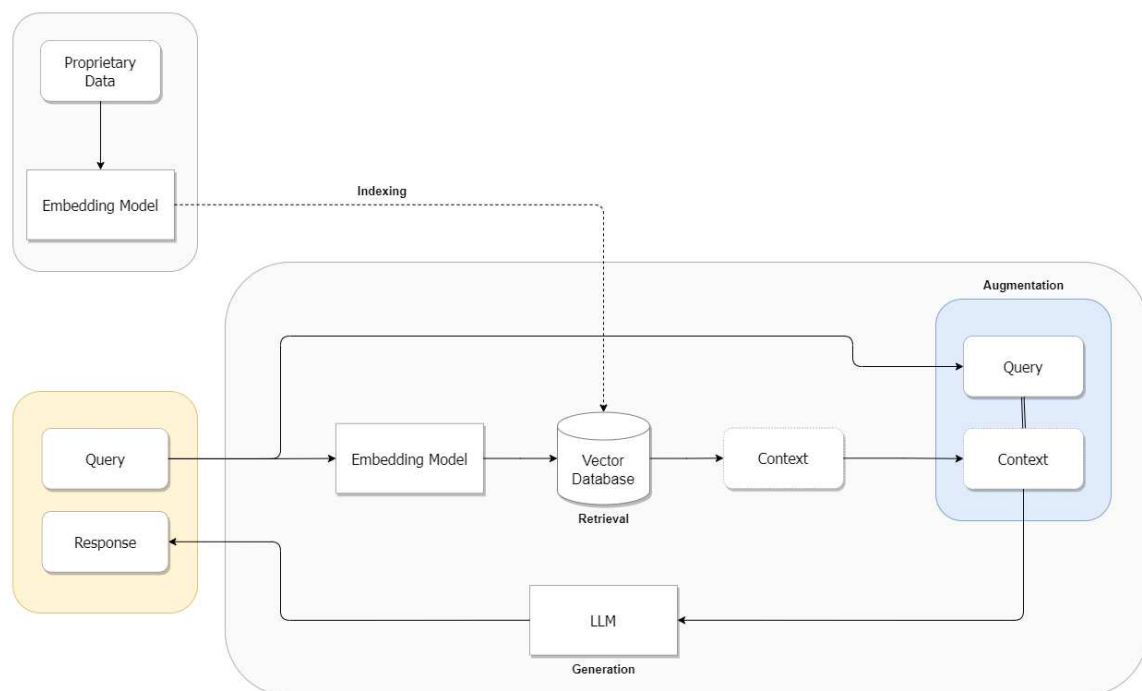


Figure 3.1: High-level view of the system architecture

core of this architecture is the vector database, which acts as a non-parametric memory, storing high-dimensional vector embeddings derived from proprietary data. This data is initially processed by an embedding model that transforms it into vectors suitable for indexing. When a user inputs a query into the system, the query is first processed by an embedding model identical to the one used for indexing the proprietary data. This

model converts the query into a vector, which is then used to search the vector database. The search process utilizes an ANN algorithm to efficiently find vectors in the database that are most similar to the query vector. The retrieved vectors, representing the most relevant contextual information, are then converted back into textual form. This contextual information is combined with the original user query to create an enriched query, which provides a more comprehensive context for the LLM. The augmentation step, illustrated in the system architecture, involves integrating this enriched context into the LLM's input. The LLM proceeds to generate a response, with both the original query and the additional context.

3.2 System Requirements

3.2.1 Vector Database

The rise of LLMs has driven the development of vector databases and vector database management systems (VDBMSs). As of now, there are over 20 commercial solutions, all produced within the past several years [13]. This explosion of options has led to a kind of a "choice overload", making it difficult to compare various vector databases and select the "right" one. Since trying out multiple databases to find the best fit is a labor-intensive and time-consuming process, it is generally not a viable option. Standardized benchmarks such as [14] and [15] make this process easier, allowing for efficient assessment of different factors of vector databases. There is a large number of factors to consider when choosing a vector database; for this system, the following aspects were considered.

Openness

One of the first decisions to make is whether to choose an open-source or a proprietary solution. For this project, an open-source database was essential due to its transparency, flexibility, and cost-effectiveness.

Language Support

The vector database should support the programming language used for the development of the system, in this case, Python. Python integration was crucial to ensure smooth and efficient development processes. Most modern vector databases support a variety of

languages, including Python, Java, and Go.

License

Since the type of license under which the database is distributed can have significant implications for its use in a project, an open-source license, such as Apache 2.0 or MIT, was essential. This type of license provides freedom to use, modify, and distribute the software with minimal restrictions.

Maturity and Community

The maturity of the database and the strength of its community were important considerations. A mature, well-established database tends to be more stable and feature-rich. Additionally, a strong community offers support and knowledge through forums, documentation, and third-party integrations.

Sparse Vectors and Hybrid Search

While sparse vectors are common in many real-world applications, and hybrid search capabilities can enhance retrieval performance by combining vector-based and keyword-based search, this was not a necessary system requirement. Given the nature of source data, hybrid search is not prioritized.

Cost

Open-source solutions generally have lower initial costs compared to proprietary ones, which was a key consideration. However, the total cost of ownership must also be considered, including deployment, maintenance, and scaling. It is important that the chosen database is not only affordable initially but also cost-effective in the long run.

3.2.2 Large Language Model (LLM)

Large language models are evolving at a remarkable rate, with new and improved models frequently released. This pace of development presents both various opportunities and challenges for leveraging them in RAG systems. With numerous solutions available, just as with vector databases, choosing the right LLM is not a straightforward task. While more parameters generally correlate with better performance and higher accuracy,

research shows that the size of the LLM is not a crucial factor. RAG systems provide significantly better results, regardless of the model's size. Therefore, the primary aim is to choose a cost-effective model that still delivers good performance. For the development of this system, the following aspects were considered.

Self-Hosting Capability

Given the private and sensitive nature of the source data, it was crucial that the large language model could be self-hosted. This requirement is critical to ensure data privacy and security, as it avoids the use of online LLM services like ChatGPT, which involve sending data to remote servers.

Model Size

Given the private and sensitive nature of the source data, it was crucial that the large language model could be self-hosted. This requirement is critical to ensure data privacy and security, as it avoids the use of online LLM services like ChatGPT, which involve sending data to remote servers.

Cost

The cost of deploying and maintaining the LLM is a significant factor in building RAG systems. This consideration generally includes hardware requirements, energy consumption, and any potential licensing fees associated with the LLM.

GDPR Compliance and Licensing

While not immediately critical, compliance with GDPR and appropriate licensing are important considerations for future scalability and legal compliance. The chosen LLM must have clear and permissible licensing terms that would not interfere with its use in commercial applications and should ideally support GDPR compliance to protect user data privacy.

3.2.3 Embedding Model

As mentioned, the embedding model's primary responsibility is to transform input text data into a high-dimensional vector space where semantically similar items are located

near each other. In contrast to the solid requirements for the vector database and the large language model, the specifications for the embedding model can be considered more flexible. First, different applications may prioritize different aspects of embedding performance. For instance, some applications may prioritize precision and recall, while others may emphasize computational efficiency or memory usage. This diversity in requirements means that there is no one-size-fits-all approach to choosing an embedding model. Second, embedding models can be tailored to specific types of data. The choice of the model varies based on the nature of the input data and the domain. Measuring the performance of an embedding model is challenging "in advance" due to its dependence on the specific context and application; the effectiveness of an embedding model is often assessed through its impact on the overall system performance in real-world tasks. Key performance metrics for embedding models include semantic similarity, retrieval accuracy, computational efficiency, and scalability. One useful resource for evaluating and selecting embedding models is the Massive Text Embedding Benchmark (MTEB) leaderboard [16]. This benchmark provides a comprehensive evaluation of various embedding models across multiple tasks and datasets, offering insights into their performance in different scenarios. In the context of RAG systems, the 'Retrieval' aspect of MTEB is particularly important. The retrieval score on MTEB assesses how well an embedding model can fetch relevant documents from a large corpus, which directly correlates to the embedding model's utility in a RAG system. Models with high retrieval scores are typically better suited for applications where the accuracy and relevance of retrieved information are critical.

3.3 Components

3.3.1 Qdrant

Taking all of the vector database requirements into consideration, Qdrant [17] was identified as the best candidate to be used as the vector database for the RAG system. Firstly, Qdrant is an open-source solution that provides flexibility and security. Its licensing, Apache 2.0, enables customization and adjustments to meet the specific requirements of the project. In addition, choosing to self-host gives control over security measures, which is important when dealing with sensitive data in the RAG framework. When it

comes to compatibility, Qdrant integrates with Python, enabling effective deployment. Despite being a relatively new database, it has rapidly established itself as a reliable and stable option with an active community providing valuable resources like thorough documentation and blogs. Cost efficiency was also a crucial factor, and Qdrant offers the best-estimated pricing.

By its own definition, Qdrant is “a vector similarity search engine that provides a production-ready service with a convenient API to store, search, and manage points (i.e. vectors) with an additional payload” [18]. At a high level, Qdrant’s architecture consists of several

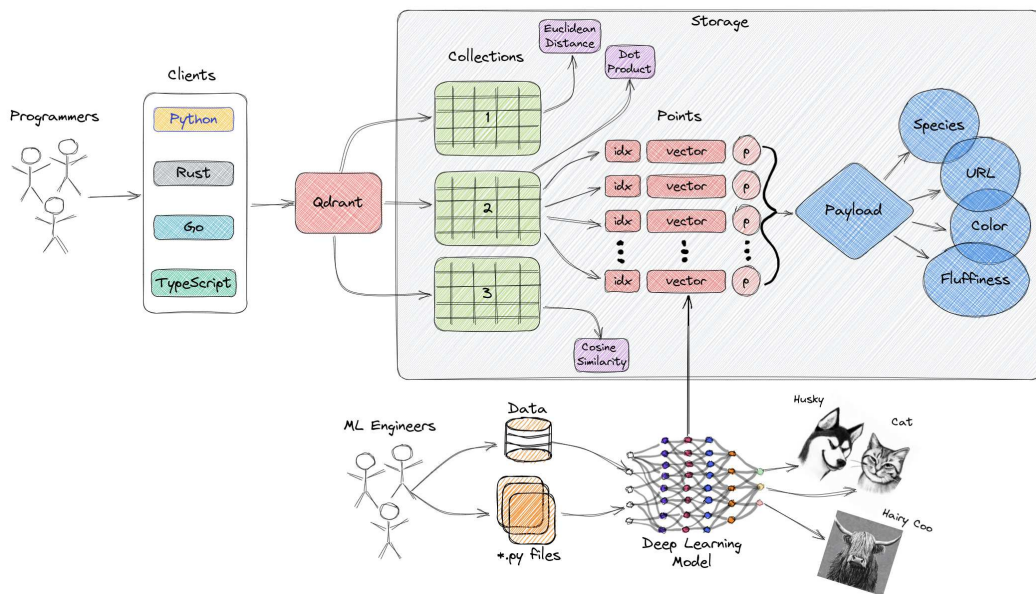


Figure 3.2: Qdrant Architecture [18]

key components. The core entities that Qdrant operates with are *points*. Each point represents a vector with optional *payload*, which can store any additional information about the vector in a JSON format. This allows for adding filters during the search, enabling the implementation of custom logic on top of semantic similarity. Points are organized into *collections*. A collection is a named set of points, where the vector of each point within the collection must have the same dimensionality and be compared by a single metric [19]. Collections allow users to manage and query related vectors efficiently by providing a way to segment data based on different criteria, such as different datasets, application contexts, or user-defined categories. As mentioned, points must be compared by a distance metric to measure similarities between them. Qdrant supports four types of metrics: dot product, cosine similarity, euclidean distance, and Manhattan distance. In terms of similarity search and vector indexing, Qdrant currently only uses HNSW as a

dense vector index. Several factors contribute to the selection of HNSW. Firstly, HNSW works well with the modification which enables Qdrant to apply filters while searching. Secondly, it ranks high in accuracy and speed when compared to public benchmarks [20, 21]. Depending on the requirements of the application, Qdrant offers flexible data storage options, allowing users to choose between maximizing search speed and minimizing RAM usage. Qdrant provides two primary storage methods: in-memory storage and memory-mapped (mmap) storage. In-memory storage keeps all vectors directly in RAM, ensuring the highest possible search speed since accessing data from RAM is significantly faster than reading from disk. This method is ideal for applications that require ultra-fast retrieval times and can allocate sufficient RAM to accommodate the entire dataset. By minimizing disk access, in-memory storage ensures that query operations are executed with minimal latency, making it suitable for performance-critical applications. On the other hand, mmap storage creates a virtual address space associated with a file on disk. Unlike traditional disk storage, mmap files are not fully loaded into RAM; instead, they utilize the operating system's page cache to access file contents on demand. This approach provides a flexible use of available memory, offering a balance between speed and memory efficiency. This makes it a good option for handling large datasets where it is impractical to fit all data in memory but still requires relatively fast access times.

3.3.2 Mistral 7B

Considering the key requirements for selecting an LLM, the Mistral AI 7B model was chosen for the RAG system. As mentioned, the ability to self-host the model was crucial due to the sensitive and proprietary nature of the knowledge base; Mistral AI offers a self-hosted solution, ensuring that no data needs to be sent to external servers, maintaining privacy and security. In terms of model size, the Mistral AI 7B model seemed to offer a good balance between complexity and resource efficiency; with 7 billion parameters, it is sophisticated enough to generate high-quality responses while still being manageable in terms of computational requirements.

In the rapidly evolving domain of Natural Language Processing (NLP), the pursuit of higher model performance often necessitates an increase in model size [22]. However, this scaling tends to raise computational costs and inference latency, thereby creating sig-

nificant barriers to deployment in practical, real-world scenarios. Designed with this in mind, Mistral 7B model delivers high performance while still maintaining efficient inference [22]. Mistral 7B outperforms the previous best, Llama 2 13B, across all tested bench-

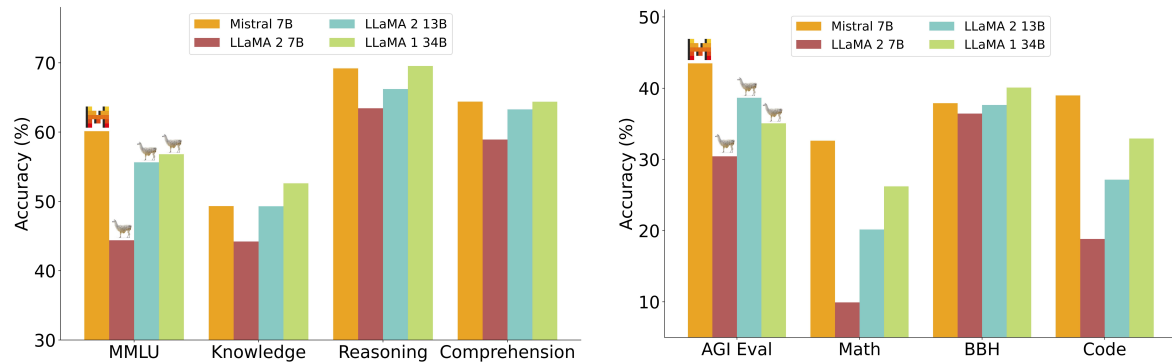


Figure 3.3: Performance of Mistral 7B and different Llama models on a wide range of benchmarks. [22]

marks. This can mostly be attributed to its architecture, which incorporates grouped-query attention (GQA) and sliding window attention (SWA). GQA significantly accelerates inference speed and reduces memory requirements during decoding, allowing for bigger batch sizes and consequently a greater throughput. Additionally, SWA is designed to handle longer sequences more effectively at a reduced computational cost. Mistral 7B is available under the Apache 2.0 license, which allows for wide-ranging use and modification. Furthermore, its integration with HuggingFace facilitates ease of deployment, enabling the implementation of the model both locally and on any cloud platform.

3.3.3 Sentence-transformers model: all-mpnet-base-v2

The selection of the embedding model was guided primarily by the model’s performance on the Massive Text Embedding Benchmark, along with the consideration of computational efficiency and system compatibility. The `all-mpnet-base-v2`, part of the Sentence-transformers model family, is specifically designed for mapping sentences and paragraphs to a 768-dimensional dense vector space and is best used for tasks such as clustering and semantic search [23]. The model is based on the MPNet architecture, which combines the advantages of BERT and XLNet while avoiding their limitations [24]. Specifically, the pre-trained Microsoft/mpnet-base model was utilized and fine-tuned on a dataset comprising 1 billion sentence pairs. The approach employs a contrastive learning objective, where the model learns to distinguish between true sentence pairs and randomly

sampled unrelated sentences [23]. Considering all that, it showed high performance in retrieval tasks (on the MTEB), which are critical for the RAG system. It is important to note that while the MTEB was an important guide in the selection of the model, the ranking of models on this benchmark can change as new models are introduced and existing models are updated. Another factor in selecting the embedding model was computational efficiency. The `all-mpnet-base-v2` model is designed to be lightweight and fast, making it suitable for the system's deployment environment, while still maintaining high accuracy and quality of the embeddings.

4 Implementation

Following the proposed architecture in 3.1, the figure 4.8 illustrates the final implementation which integrates multiple components and services deployed on Google Cloud Platform (GCP). The process begins with the Confluence data, which is loaded and pro-

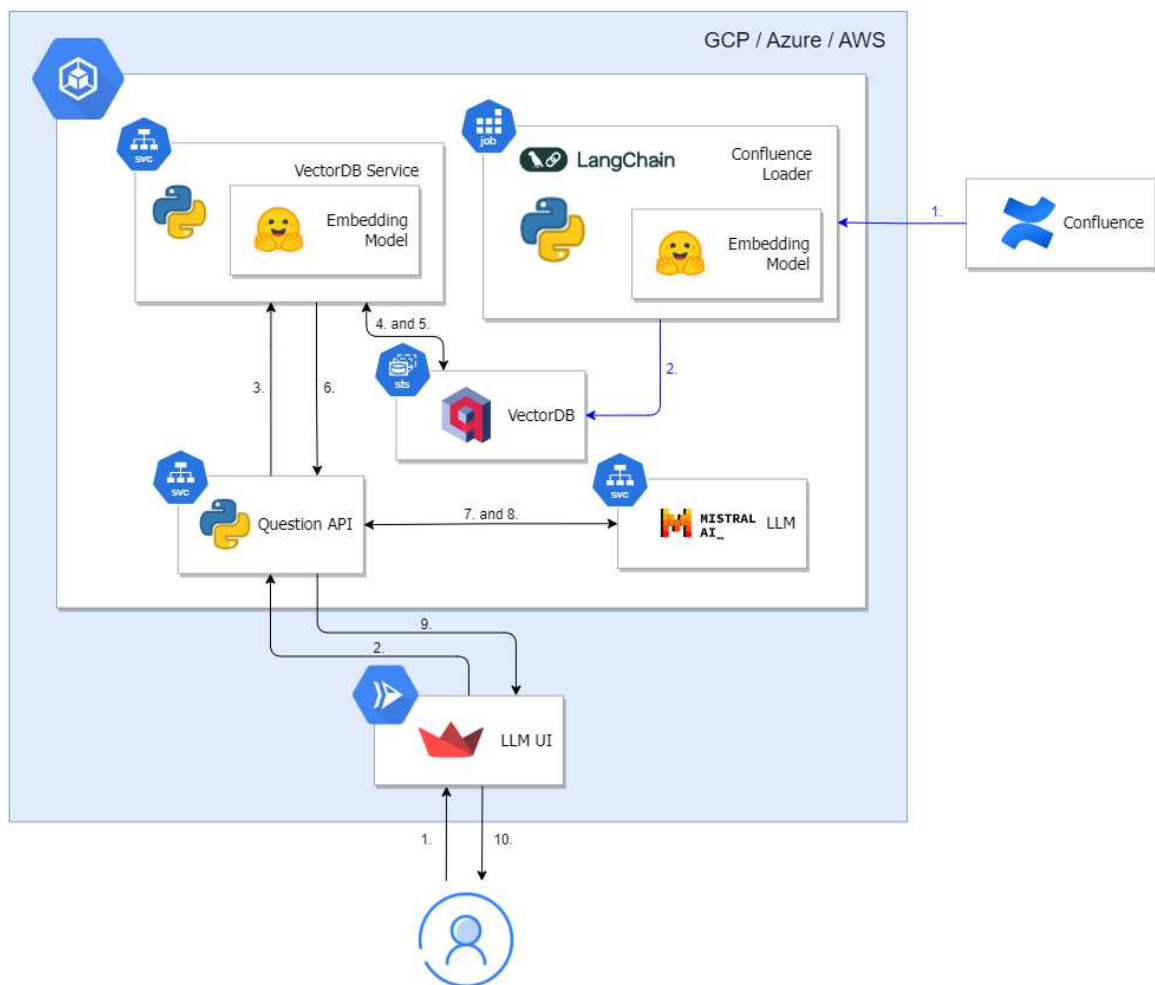


Figure 4.1: System Architecture

cessed by the system. This initial step (Step 1) involves the Confluence Loader, a component responsible for extracting data from the Confluence knowledge base. The extracted

data is then fed into an embedding model, which transforms the text data into high-dimensional vectors. In Step 2, the embeddings generated from the Confluence data are stored in the Qdrant vector database. When a user submits a query through the LLM UI (Step 1 for user query input), it triggers the Question API (Step 2). The Question API is a service that processes the incoming query. The query is then passed to the VectorDB Service (Step 3), where the same embedding model converts the query into a vector. This query vector is used to search Qdrant (Step 4) for relevant documents. Qdrant retrieves the closest matching embeddings based on semantic similarity (Step 5). The relevant embeddings retrieved from Qdrant are then sent back to the Question API (Step 6). In Step 7, the context-enriched query is forwarded to the LLM, which in this case is implemented using Mistral AI's large language model. The LLM processes the query along with the context provided by the retrieved embeddings to generate a comprehensive response (Step 8). The generated response is then sent back to the Question API (Step 9), which formats it appropriately before sending it to the LLM UI (Step 10). Finally, the user receives the response, completing the pipeline.

4.1 Indexing Phase

The indexing phase is crucial for preparing documents for efficient retrieval and generation. Firstly, documents are fetched from Confluence using a custom `ConfluenceLoader` object, created for loading and formatting the pages. Once fetched and processed, documents are chunked with the help of a `DocumentChunker` object, using specific chunking methods. This process ensures that documents are divided into smaller parts, improving the retrieval performance in the database. Finally, the chunked documents are inserted into the vector database using a custom `Qdrant` object.

4.1.1 Document Loading

To create a knowledge base, the relevant proprietary data must first be fetched. As mentioned, all of the data to be extracted is stored on Confluence, in the form of Confluence pages. The general architecture of Confluence revolves around the concept of spaces, which serve as high-level containers for organizing content. Within each space, pages are structured hierarchically, allowing users to create parent and child pages to organize

information logically. Each page within Confluence can contain a variety of content types, including text, images, tables, attachments, and macros.

For the purpose of loading data from Confluence pages, a Document Loader component, specifically Langchain's `ConfluenceLoader` is used. Acting as an interface to the Confluence API, it uses the `URL`, `username`, and `api_key` parameters to connect to the right Confluence instance. The `keep_markdown_format` parameter is determined based on the specified chunking strategy. If the chunking strategy is set to "markdown," this parameter is set to `True`, indicating that the content should retain its Markdown formatting during extraction. This is needed in the next step if a special markdown chunker is used. If the chunking strategy is not "markdown," this parameter is set to `False`, and the content is extracted in a plain text format. The `load()` method of the `ConfluenceLoader`

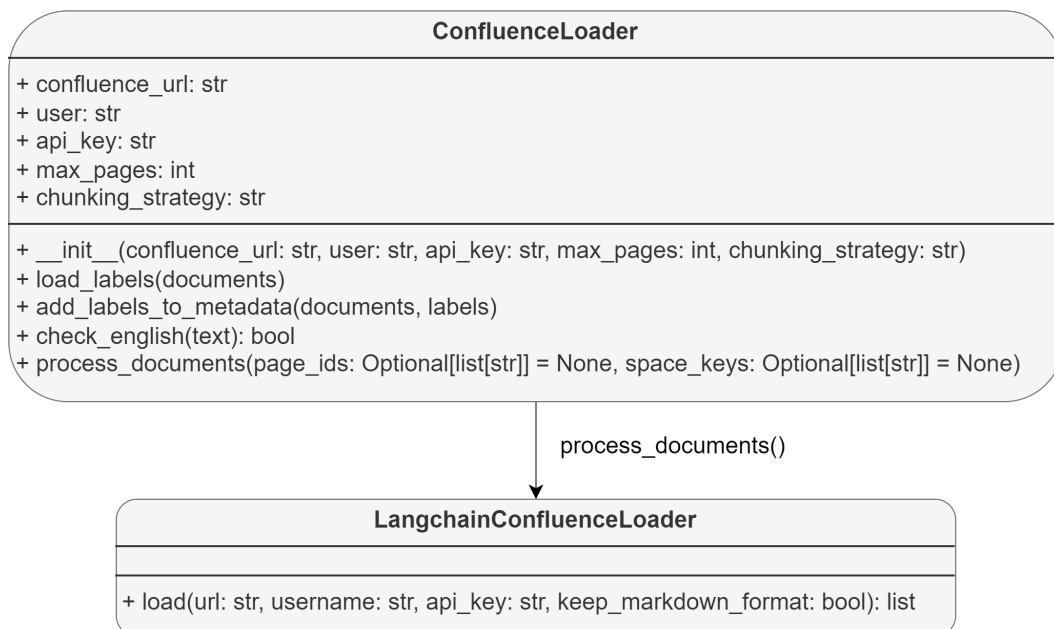


Figure 4.2: Diagram of a `ConfluenceLoader` class

class offers several parameters to configure the retrieval process. Firstly, the `limit` parameter determines the number of pages fetched per request, with a default value of 50. It is important to note that the Confluence API, which is internally used by the `load()` method, caps the responses to 100 pages per request, so even if a higher number is set, 100 pages will be loaded per request. Additionally, the `max_pages` parameter in the same method sets the maximum total pages to retrieve, defaulting to 1000. Furthermore, the `include_attachments` parameter, a boolean value, dictates whether attachments associated with Confluence pages should also be loaded. When enabled

(True), all supported attachment types, including PDFs, images (PNG, JPEG/JPG, SVG), and document files (Word, Excel), are downloaded. ConfluenceReader then extracts text from these attachments and appends it to the corresponding Document object.

Originally designed to load pages from a single Confluence space, the ConfluenceLoader has been augmented with custom logic to extend its functionality, enabling the loading of pages from multiple spaces. Along with the page content, ConfluenceLoader loads additional metadata for each document. The metadata consists of the page title, page ID, and the address of the Confluence page. Since Confluence offers the functionality of labels - metadata tags that users can assign to pages to serve as keywords for easier categorization and enhancement of content navigation and discoverability, we wanted to store the page labels as metadata. As a result, custom methods were created: the first method retrieves labels associated with Confluence pages; it takes a list of documents as input, extracts their unique identifiers, and then queries the Confluence API to fetch the labels for each document. The retrieved labels are stored in a dictionary where the keys correspond to the document IDs. The second method incorporates the retrieved labels into a new field *label* to the metadata of each Document object. Another custom

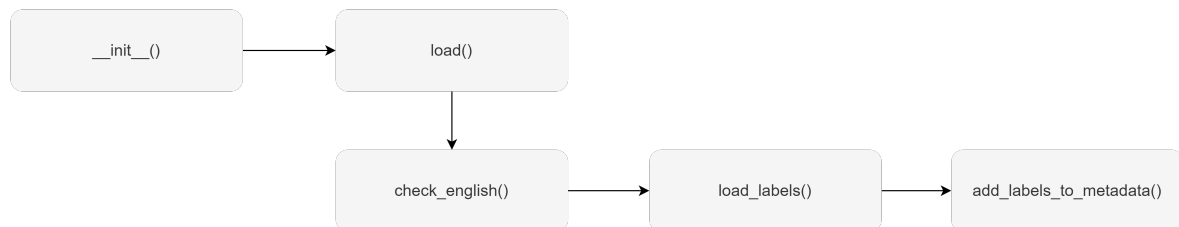


Figure 4.3: Diagram of a ConfluenceLoader class

functionality that was implemented was the filtering of Confluence pages in languages other than English (Croatian in this case). The choice to have only English documents in the knowledge base is based on multiple factors. The Large Language Model (LLM) used does not have a natural proficiency in Croatian, resulting in limited or non-existent capability to understand and produce Croatian text. With this restriction in place, adding Croatian documents may result in errors or misunderstandings in the database, impacting the accuracy and dependability of the information produced or obtained by the system. Furthermore, in the context of similarity search, it is important for pages in Croatian to be searched in Croatian in order to achieve the best search results. However, the predominant language of the proprietary knowledge base is English. As a result, to

maintain consistency in language and ensure optimal search results, the pages in Croatian are filtered out and not stored in the vector database.

4.1.2 Text Chunking

Once documents are loaded from the corpus, it is essential to split them into smaller parts - chunks. The primary goal of chunking is to improve the relevance of the information retrieved from the similarity search in the vector database. Implementing a successful chunking method guarantees that the search results capture the meaning and context of a user's query.

Choosing an optimal chunking strategy is not a straightforward task, as several factors within the RAG pipeline must be considered. The first factor to consider is the structure of the indexed content; documents with clear structure may benefit from specialized chunking approaches. Another component to consider is the embedding model used; factors like the model's context input length influence the chunking process, with certain models exhibiting better performance on chunks of specific lengths. As LLM stands at the end of the RAG pipeline, it is important to consider token limits. Since LLMs have limited context windows, the size of the chunk impacts the amount of context that can be input into the LLM, with larger chunks imposing constraints on the top-k retrieval mechanism.

Chunking strategies can be broadly categorized into two main types: *token-based* and *semantic*. The first category, token-based chunking, encompasses several approaches such as character chunking, sentence splitting, and specialized chunking. They involve breaking down text into smaller segments based on predefined tokens or units of meaning, such as words or characters. Generally, token-based splitters have two fundamental parameters: `chunk_size` and `chunk_overlap`. The `chunk_size` parameter sets a limit on the maximum size allowed for each chunk, guaranteeing that the resulting segments are a manageable length, while the `chunk_overlap` parameter helps maintain the semantic context between consecutive chunks by determining the amount of overlap between them.

The second category, semantic chunking, focuses on the meaning of chunks within the document rather than adhering to a globally set chunk size. This approach involves creating embeddings for each sentence, comparing similarities, and grouping the most similar

ones together. While semantic chunking captures the semantic meaning and context of documents, it comes with a trade-off of increased computational complexity compared to token-based methods.

Character Splitter

Character text splitters represent the most straightforward approach to chunking text data. Unlike more complex methods relying on Natural Language Processing (NLP) libraries, character splitters work by directly segmenting text based solely on characters. However, Langchain's `CharacterTextSplitter` also has a `separator` parameter which specifies the separator chunks are split on, while the `chunk_size` parameter determines the maximum number of characters allowed in each chunk.

Sentence Splitter

Sentence splitters, as opposed to character splitters, take a more linguistically informed approach by focusing on dividing text into meaningful linguistic units - sentences. Sentence splitters use NLP methods to find sentence boundaries using grammar, punctuation, and context clues. There are several sentence splitter implementations in Langchain, including the `NLTKTextSplitter` and `SpacyTextSplitter`. The `NLTKTextSplitter` utilizes the NLTK package, specifically its `sent_tokenize` function [25], to tokenize the text into a list of sentences, which are later merged into chunks. Upon initialization, it allows customization of parameters such as the separator between chunks and the language for tokenization. On the other hand, `SpacyTextSplitter` leverages the Spacy package to split text. The initialization parameters include the pipeline (defaulting to "en_core_web_sm") [26] and the maximum length of characters the Spacy model can handle. During the `split_text` method, `SpacyTextSplitter` tokenizes the text using the chosen Spacy pipeline and iterates over the generated sentences, collecting them into chunks and merging them.

Recursive Splitter

Recursive chunking is a method of dividing input text into smaller chunks in a hierarchical and iterative manner. This approach uses a specified list of separators to determine where to split the text. Initially, the text is divided using the first separator in the list.

If the resulting chunks are not of the desired size, the method recursively calls itself on these smaller chunks using a different separator from the list. This process continues until the final chunk size is achieved. While the resulting chunks may not be exactly the same size, they strive to be of a similar size, maintaining a balance between granularity and coherence. For Langchain's `RecursiveCharacterTextSplitter`, the default list of separators is `["\n\n", "\n", " ", ""]`; this attempts to ensure that paragraphs, sentences, and words remain together as much as possible since they generally have the strongest semantic relation [27].

Document Specific Splitter

Document-based splitters take the inherent structure of the document into account. Rather than using a set number of characters or a recursive process, a document-based splitter creates chunks that align with the logical sections of the document, like paragraphs or subsections [28]. This approach proves particularly beneficial when dealing with Markdown files containing hierarchical information, code snippets, or enumerated lists, as it ensures that each chunk retains the semantic coherence and formatting integrity of the original Markdown content. Langchain offers a `MarkdownHeaderTextSplitter` which splits the markdown document on headers.

To have flexibility in splitting the documents, a custom `DocumentChunker` class was implemented. At initialization, the class accepts a strategy parameter, indicating the chosen text-splitting approach, along with optional keyword arguments for chunk size and the overlap between them. The core functionality of the `DocumentChunker` class lies in the `chunk_documents()` method, which takes a list of documents as input and applies the specified text-splitting strategy to partition each document into smaller chunks. Based on the chosen strategy, the method instantiates an appropriate text splitter object, such as `RecursiveCharacterTextSplitter`, `SpacyTextSplitter`, `NLTKTextSplitter`, or `MarkdownTextSplitter`, passing any relevant keyword arguments to configure the splitter's behavior. In cases where the input documents are in Markdown format ("markdown" strategy), the method employs a two-step approach. First, `MarkdownTextSplitter` is used to segment the Markdown documents based on set headers. Subsequently, the `RecursiveCharacterTextSplitter` is applied to further divide the resulting chunks into smaller units based on character boundaries.

4.1.3 Embedding and Storing

After completing the chunking step, the generated document chunks need to be embedded for efficient storage and retrieval within the system. The initial step in this process involves selecting an appropriate embedding model. A good source for comparing and choosing such models is the MTEB Leaderboard on HuggingFace [16] since it is the most up-to-date list of proprietary and open-source text embedding models, accompanied by statistics on how each model performs on various embedding tasks such as retrieval, summarization, etc. The “retrieval” benchmark is particularly important for our system since it evaluates the ability of the embedding model to retrieve relevant content in asymmetric search scenarios – specifically, matching short queries with longer texts [29].

For the process of embedding and storing data chunks, the `Qdrant` class is implemented. The embedding process begins with initializing the embedding model that will convert text documents into vector representations. In the `Qdrant` class, this is handled by the `get_embedding_model()` method. This method takes the name of the desired model, specified by the `embedding_model_name` parameter, and initializes it using the `HuggingFaceEmbeddings` class. The name of the model corresponds to a pre-trained model available on HuggingFace’s Model Hub. When initialized, it triggers the download of the specified model from HuggingFace’s servers. Once the embedding model is initialized, it runs locally on the specified device. In the `Qdrant` class example, the model is configured to run on a CPU. This configuration is determined by the `model_kwargs` parameter passed during initialization.

The embedding and storing is encapsulated inside the `insert_documents_into_collection` method. Here, the key component is Langchain’s `from_documents()` method, which establishes a connection to the `Qdrant` server using the provided URL and port, and it creates or updates the specified collection with the new embeddings. The `collection_name` parameter specifies the target collection, and the `force_recreate` parameter determines whether the collection should be recreated if it already exists.

4.2 Generative Phase

After the indexing phase, where all relevant data is processed and stored in a vector knowledge base, the generative phase can begin. This phase bridges the gap between

the retrieved information and the final output presented to the user. It begins by embedding the user’s input query and performing a similarity search, identifying content that can provide context for the user’s question. At its most basic level, the retrieval process involves a top-k vector retrieval, which typically returns the 3-10 most similar matches to the user’s query. Ideally, the system would find a single passage that directly answers the query. However, because information often overlaps and is rich in context, it is necessary to sample multiple text segments. This ensures that the LLM has sufficient context to generate an answer [29]. The retrieved context is then fed into the LLM along with the initial question. The generative phase is designed around three core services: the user interface (UI), the VectorDB Service, and Question API Service.

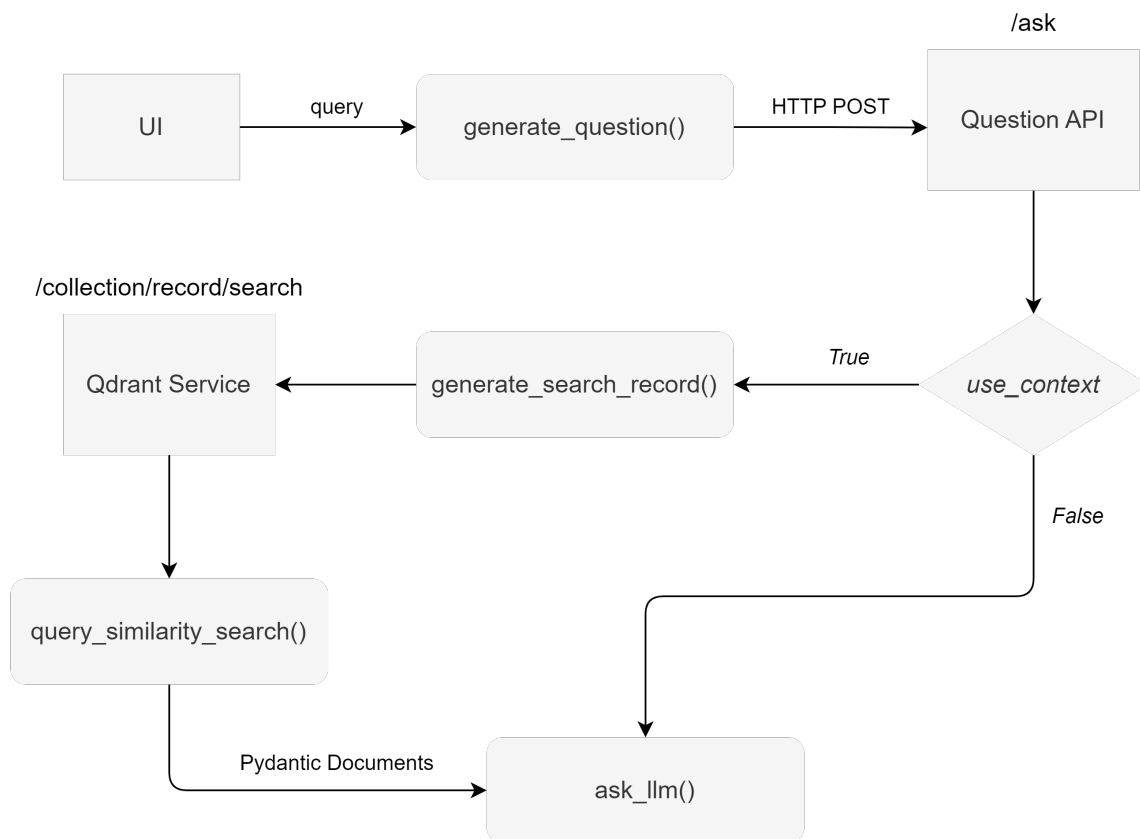


Figure 4.4: System Architecture

4.2.1 Retrieval

In the retrieval phase of the system, the user’s query is processed to retrieve relevant information from the vector database. This process begins with an HTTP POST request being sent from the UI component to the /ask endpoint of the Question API service,

facilitated by Python's requests library. The payload of this request consists of a JSON object constructed by the `generate_question()` method, which takes the user's input message and a boolean flag indicating whether context from the RAG model should be utilized in the generation phase and creates a `AskQuestionRequest` object.

So, the Question API service's `/ask` endpoint expects an `AskQuestionRequest` object in the request body, encapsulating the user's question and a flag indicating if context from the knowledge base should be used for the generation by LLM. If the `use_context` flag is set to `True`, the service generates a `SearchRecord` object based on the user's question. This search record is subsequently sent to the Qdrant Service, specified by the `qdrant_service_endpoint`, to retrieve similar records that may provide context for the question. In the case the `use_context` flag is `False`, the retrieval phase is bypassed, and the user's query is directly forwarded to the LLM.

The Qdrant Service features a route at `/collection/record/search`, which handles requests to search for records based on a provided `SearchRecord` object. Upon receiving a search request, the service invokes the `query_similarity_search` method of the Qdrant instance, passing the input query and limit as parameters. This method executes a similarity search in the database, leveraging the provided query, and returns a list of `PydanticDocument` objects representing the search results.

Data Models

For communication between the services, several Pydantic models are created. Pydantic, a data validation library in Python that enforces type hints at runtime and provides serialization and deserialization capabilities, ensures correct and consistent data exchange between FastAPI services. The following models are used:

SearchRecord

The `SearchRecord` model defines the structure of a request for similarity search in the vector database based on the user's input query, an optional limit on the number of results, and optional filters for refining the search.

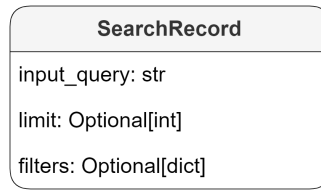


Figure 4.5: SearchRecord Data Model

AskQuestionRequest

The AskQuestionRequest model specifies the format of requests to ask questions, comprising the user's input query and a boolean flag indicating whether context from the RAG model should be utilized.

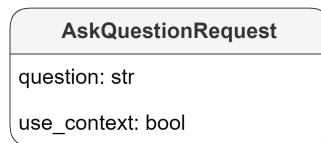


Figure 4.6: AskQuestionRequest Data Model

AskQuestionResponse

The AskQuestionResponse model outlines the structure of responses to question requests, containing the answer to the question and an optional list of sources - Confluence pages which were used as context to generate the answer.

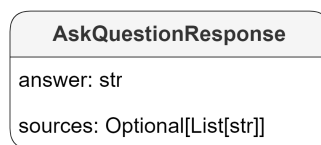


Figure 4.7: AskQuestionResponse Data Model

PydanticDocument

The PydanticDocument model describes the format of documents retrieved from the vector database, including the page content and optional metadata associated with the document.

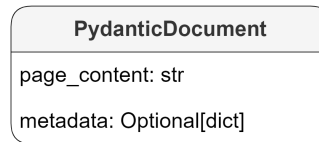


Figure 4.8: PydanticDocument Data Model

4.2.2 Generation

After retrieving similar top k (specified by the `limit` parameter of the `SearchRecord` object) similar records from the Qdrant Service, the documents are passed to the `ask_llm()` method, which serves as an interface to interact with the LLM. The method takes two parameters: `question` which is a string representing the user's query, and `similar_records` which is a list of `PydanticDocument` objects, representing the retrieved documents. These documents serve as a context for generating a response. Inside the method, the `get_selected_llm()` method is responsible for retrieving the appropriate LLM, set as an environment variable. This function takes two parameters: the `llm_type` indicating which LLM is used, and the `llm_url`, representing the endpoint URL of the deployed language model. Based on the provided LLM type, the function returns an instance of the corresponding LLM class. Upon instantiation of the correct LLM class instance, the `ask()` method is invoked. This abstract method is responsible for constructing a prompt for the LLM model based on the input question and context documents.

The method starts by initializing an empty string variable `prompt`. If context documents are provided, it iterates over a list of `PydanticDocument` objects to extract the title and page content from the document's metadata and append it to a `context_str` string, ensuring each document's content is separated by double newlines. After constructing the context string, it combines this context with the user's question to form a prompt. The context and question are enclosed within a special instruction format:

```
[INST] Context: <context_str> Given the above context and your general
knowledge, answer the question:<question>[/INST]
```

Large Language Model Parameters

The `ask()` method is an abstract method of a LLM class, providing a blueprint that should be implemented by any subclasses.

The `Mistral7B` class extends the `LLM` class and provides a specific implementation of the `ask` method, mainly by setting specific parameters for the language model. The

Parameter	Description	Value Used
<code>max_new_tokens</code>	Maximum number of tokens that the model can generate in response to the prompt.	1000
<code>top_k</code>	Restricts the sampling pool to the top-k highest probability tokens at each step.	50
<code>temperature</code>	Controls the randomness of the model's predictions.	0.001
<code>top_p</code>	Sets a cumulative probability threshold for nucleus sampling.	0.95
<code>num_return_sequences</code>	Specifies the number of response sequences to generate.	1

Table 4.1: LLM Parameters

`max_new_tokens` parameter sets the maximum number of tokens that the model can generate in response to the prompt. By limiting the number of new tokens, the length of the generated output is controlled, preventing overly long responses. In this implementation, `max_new_tokens` is set to 1000, allowing for detailed and comprehensive answers while maintaining a reasonable response length. The `top_k` parameter restricts the sampling pool to the top-k highest probability tokens at each step in the text generation process. This approach makes the output more deterministic by narrowing the model's focus to the most likely tokens. A `top_k` value of 50 means that out of all possible tokens, only the 50 tokens with the highest probabilities are considered. The `temperature` parameter controls the randomness of the model's predictions. A lower temperature results in less random and more deterministic outputs, while a higher temperature increases randomness and creativity. In this case, the temperature is set to 0.001, making the responses very deterministic and focused on high-probability tokens, making the model rely almost solely on the context provided by the knowledge base. The `top_p` parameter, also known as nucleus sampling, sets a cumulative probability threshold. Instead of considering a fixed number of top tokens (as in `top_k`), it includes the smallest set of tokens whose cumulative probability exceeds top-p. For example, with `top_p` set to 0.95, the model considers tokens until their cumulative probability is at least 95%. This allows for dynamic adjustment of the sampling pool size, providing a balance between diversity and coherence. Using both parameters helps the model introduce some creativity while still

focusing on the most probable tokens. Finally, the `num_return_sequences` parameter specifies the number of response sequences to generate. Setting it to 1 means that only a single response is generated for each input prompt. This is appropriate for scenarios where a single, best response is desired rather than multiple alternative responses.

4.3 User Interface

For the development of the system's user interface, the Streamlit Python library was used. Streamlit offers several Chat elements [30], providing a simple and straightforward method for creating UIs for chatbots. The `st.text_input` component is used to

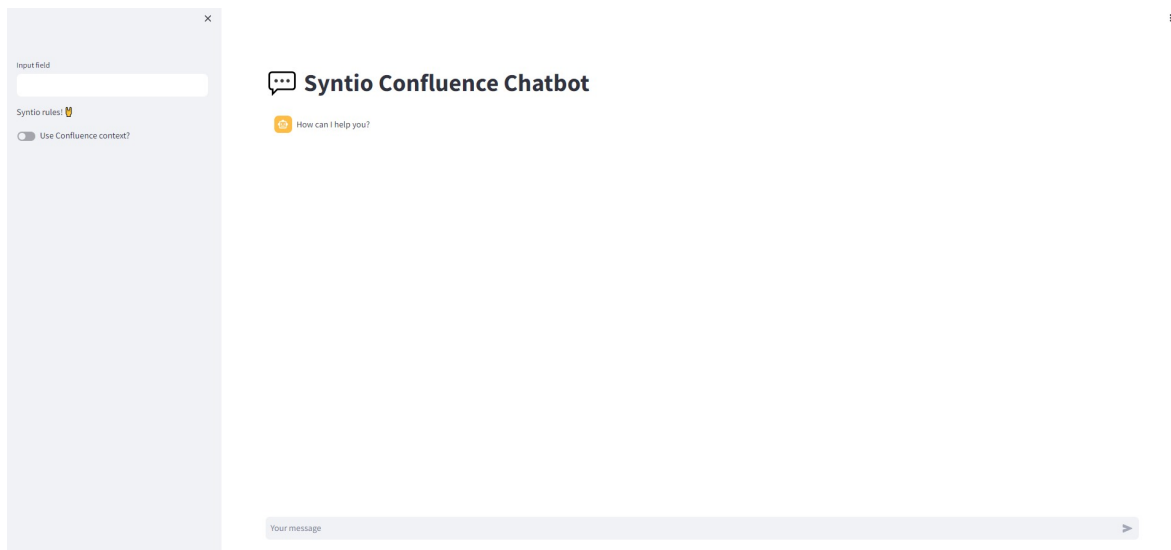


Figure 4.9: User Interface for the RAG-based LLM system

capture user input text. Additionally, a sidebar widget, created with `st.sidebar`, includes a toggle switch (`st.toggle`) that allows users to decide whether to use context from Confluence in their queries. The main interface is set up with `st.title` to display the chatbot's title, and a session state is maintained to store messages exchanged during the chat. The message flow is handled by iterating over `st.session_state.messages`, displaying each message using `st.chat_message`. When a user inputs a query through `st.chat_input`, the system updates the session state, sends the query to the Question API using an HTTP POST request via the `requests` library, and processes the response. The response, including any referenced sources, is then displayed back to the user.

4.4 Deployment

The deployment process of the RAG system involves building Docker images for each service and pushing them to the Google Artifact Registry. The core services are deployed on a GKE cluster, while the user interface is deployed using Google Cloud Run. Terraform scripts automate the provisioning and management of the infrastructure, ensuring a consistent and scalable deployment.

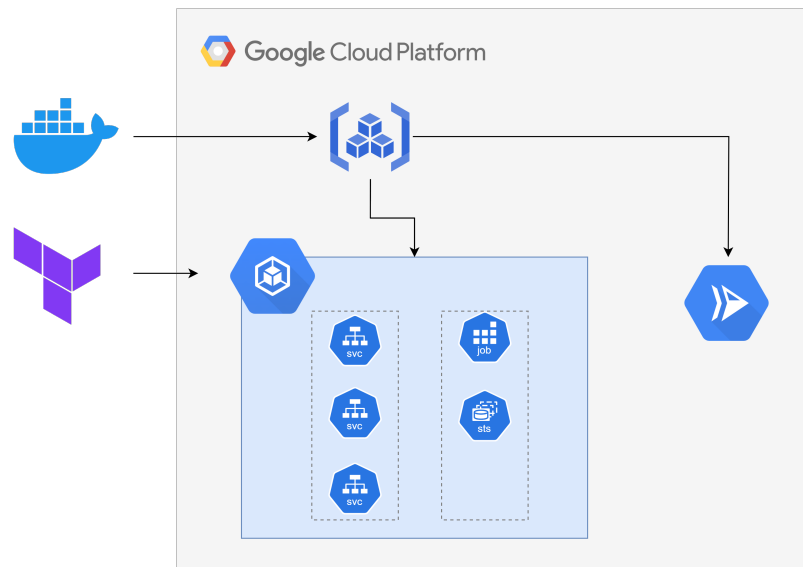


Figure 4.10: High-level overview of the deployment process

4.4.1 Google Cloud Platform

Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) is a managed Kubernetes service provided by Google Cloud Platform. It simplifies the deployment, management, and scaling of containerized applications using Kubernetes. In this project, GKE is used to host the main services of the RAG system, providing a resilient and scalable environment. The process begins with defining the Kubernetes cluster specifications, such as the number of nodes and their machine types, using Terraform scripts. These scripts automate the creation and configuration of the GKE cluster, ensuring consistency and repeatability. Once the cluster is provisioned, the next step involves deploying the containerized services onto GKE.

Google Cloud Run

Google Cloud Run is a fully managed compute platform that automatically scales stateless containers. In this project, Cloud Run is utilized to deploy the user interface (UI) of the RAG system. The choice of Cloud Run for the UI deployment is driven by its ability to handle HTTP requests efficiently and its seamless integration with other Google Cloud services. The deployment process for the UI involves building a Docker image of the UI application and pushing it to the Google Artifact Registry. Terraform scripts then configure Cloud Run to pull the image from the registry and deploy it as a service.

4.4.2 Deploying the System

Docker Images

Docker images are crucial for packaging the RAG system's services into portable and consistent units. Each service, including the retrieval component, the language model, and the UI, is containerized using Docker. The Dockerfile for each service defines the environment and dependencies required to run the service, ensuring that it behaves the same way in any environment. Once the Docker images are built, they are pushed to the Google Artifact Registry, a secure and scalable repository for storing and managing container images. This centralized repository allows for easy access and deployment of the images across the GKE cluster and Cloud Run.

Terraform

Terraform is an open-source infrastructure-as-code (IaC) tool that allows for the declarative configuration of cloud resources. In this project, Terraform scripts are used to automate the provisioning and management of the entire infrastructure, including GKE, Cloud Run, and the Google Artifact Registry. The Terraform scripts define the desired state of the infrastructure, including the Kubernetes cluster configuration, service deployments, and networking setup. When the scripts are executed, Terraform interacts with the Google Cloud API to create and configure the resources as specified. This approach ensures consistency, reduces the risk of manual errors, and enables version control of the infrastructure configuration.

5 Results

5.1 RAG Evaluation

Evaluating the quality of text generated by RAG systems presents significant challenges, largely due to the absence of standardized industry benchmarks. Often, these systems are assessed through human labeling, which involves creating a "gold standard dataset" [31, 32]. This dataset includes a set of questions, expected answers, and the expected context retrieved from the knowledge base. The RAG-based language model is then queried with these questions, and its responses are evaluated against the expected answers. However, this manual process is expensive, time-consuming, and inherently biased due to human subjectivity.

To achieve a more systematic evaluation, traditional benchmarks and metrics for question-answering (QA) systems, such as ROUGE and BLEU, are often employed. Unfortunately, these metrics have shown poor correlation with human judgment [33]. Consequently, the use of LLMs for evaluation is gaining traction. LLM-based evaluation leverages the capabilities of language models to simulate human judgment, assessing the relevance, accuracy, and overall quality of responses generated by RAG systems. However, using LLMs as evaluators is also not without challenges. It has been shown that LLMs exhibit systematic biases similar to humans. For instance, they tend to prefer their own outputs, are sensitive to the relative position of outputs, and score longer responses higher [34]. To address these limitations, frameworks like RAGAS have emerged. RAGAS primarily uses *Zero-Shot LLM Evaluations*, where a large language model is prompted with a template to rate the relevance of search results to a query on a scale of 1 to 10. Transitioning from *Zero-Shot* to *Few-Shot LLM Evaluation* involves adding a few labeled examples to the prompt, illustrating the desired output format and quality. This enhances the LLM's ability to understand evaluation criteria more effectively. To mitigate the problem

of manually creating QA datasets, RAGAS generates synthetic test datasets by creating synthetic queries from a document corpus. It ensures their quality through the *roundtrip consistency principle*, which checks if the original document can be retrieved using the synthetic query.

While this approach holds promise, implementing it for proprietary systems introduces several challenges. RAGAS uses OpenAI for synthetic dataset generation as well as for evaluating the RAG pipeline. However, when dealing with proprietary data, sending the corpus to RAGAS is not feasible due to confidentiality concerns. Although RAGAS offers a "bring your own" option for LLMs and embeddings, configuring this option to work correctly has proven difficult, as evidenced by community feedback [35] and testing. One major issue is that RAGAS does not currently support different templates for custom models. If the prompt is not in the correct format, the output of the LLM can vary and may not be parsed correctly. This limitation complicates the use of RAGAS with proprietary data, highlighting the need for further development and customization capabilities in such frameworks.

5.1.1 System Testing

The evaluation of RAG systems, as discussed in the previous section, is full of challenges. Traditional metrics are poorly correlated with human judgment; while emerging frameworks offer promising approaches for systematic evaluation, they pose difficulties when dealing with proprietary data due to confidentiality concerns and technical limitations. For the purpose of testing out this RAG system, a more flexible human evaluation approach was chosen. This was done for several reasons. Human labeling is inherently resource-intensive; creating a detailed *gold standard dataset*, where each query is paired with an expected answer and context, is time-consuming. Instead of sticking strictly to specific metrics, judgment was used to evaluate the overall relevance and utility of the responses. Since the knowledge base consisted of internal company information, this approach allowed for a more natural and comprehensive assessment. The evaluation focused on practical, real-world scenarios rather than theoretical benchmarks. The emphasis was on situations that directly mattered to the intended users of the system.

Question-Answer Examples

The RAG system was first tested on information specific to the company and its knowledge base. In Figure 5.1, the system provides step-by-step instructions for setting up a company VPN connection, along with some considerations such as ensuring the laptop is free of pirated software. These steps correlate with the Confluence pages with detailed instructions for VPN.

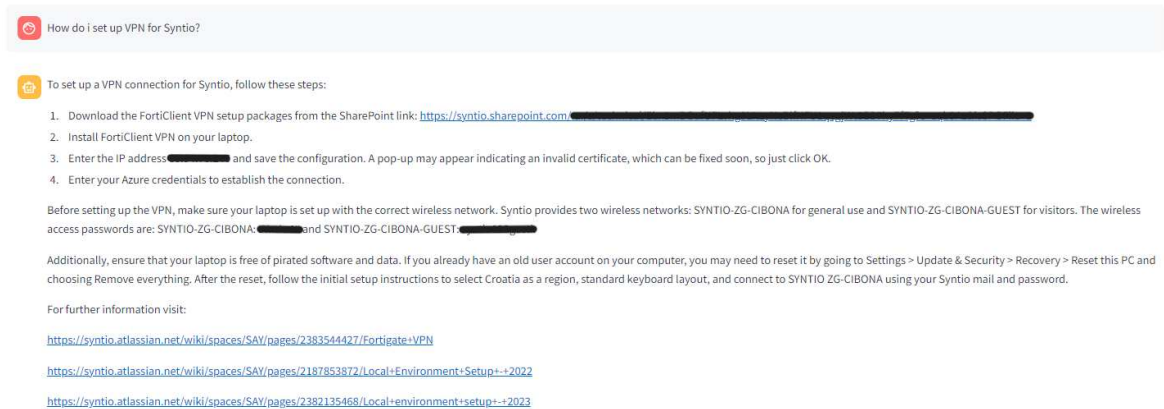


Figure 5.1: Question about VPN setup with context

In Figure 5.2, the system provides information on the focus and structure of the company's Summer School project. It mentions the specific technologies involved, such as Azure Cloud, Cloud Computing, DevOps, Containerization, Message Brokers, and Data Governance. It also outlines the activities involved, like creating and deploying Azure resources, following Git Flow for code development, and delivering presentations. Additionally, it includes information about the project's schedule and pauses between lectures.

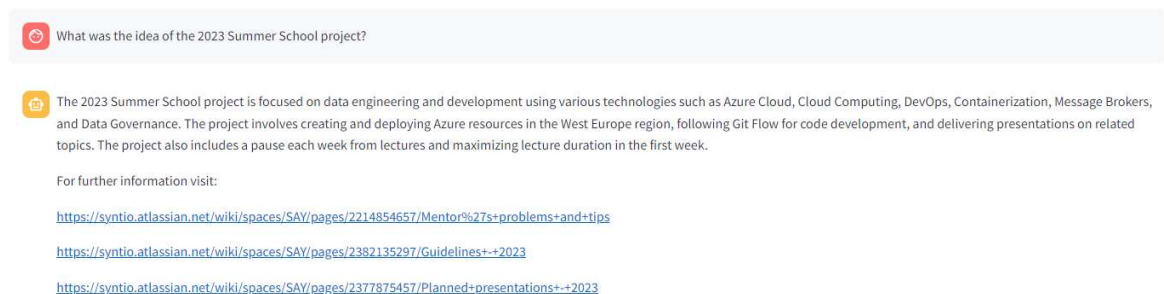
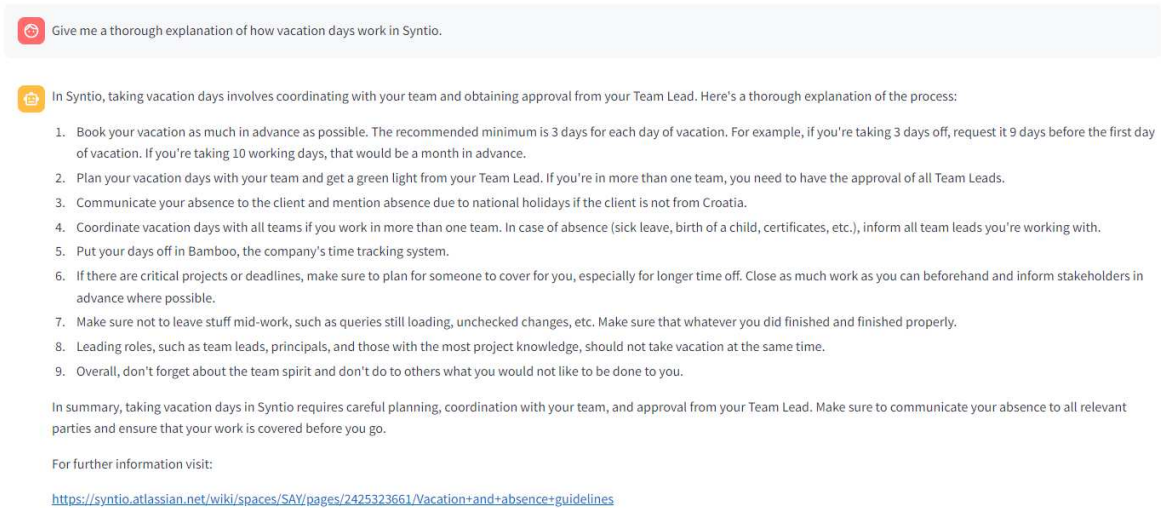


Figure 5.2: Question about 2023 Summer School Project

Figure 5.3 outlines the process for taking vacation days. It includes specific guidelines,

such as booking vacations in advance, using the company's time-tracking system (Bamboo), and ensuring that critical projects are managed.



The screenshot shows a chat interface with a light blue header containing a question: "Give me a thorough explanation of how vacation days work in Syntio." Below the question, an answer is provided, starting with "In Syntio, taking vacation days involves coordinating with your team and obtaining approval from your Team Lead. Here's a thorough explanation of the process:" followed by a numbered list of 9 steps. The steps cover booking in advance, getting team approval, communicating absence to clients, coordinating with other teams, handling critical projects, and ensuring work is covered. Below the list, there is a summary paragraph and a link to a wiki page for further information.

Give me a thorough explanation of how vacation days work in Syntio.

In Syntio, taking vacation days involves coordinating with your team and obtaining approval from your Team Lead. Here's a thorough explanation of the process:

1. Book your vacation as much in advance as possible. The recommended minimum is 3 days for each day of vacation. For example, if you're taking 3 days off, request it 9 days before the first day of vacation. If you're taking 10 working days, that would be a month in advance.
2. Plan your vacation days with your team and get a green light from your Team Lead. If you're in more than one team, you need to have the approval of all Team Leads.
3. Communicate your absence to the client and mention absence due to national holidays if the client is not from Croatia.
4. Coordinate vacation days with all teams if you work in more than one team. In case of absence (sick leave, birth of a child, certificates, etc.), inform all team leads you're working with.
5. Put your days off in Bamboo, the company's time tracking system.
6. If there are critical projects or deadlines, make sure to plan for someone to cover for you, especially for longer time off. Close as much work as you can beforehand and inform stakeholders in advance where possible.
7. Make sure not to leave stuff mid-work, such as queries still loading, unchecked changes, etc. Make sure that whatever you did finished and finished properly.
8. Leading roles, such as team leads, principals, and those with the most project knowledge, should not take vacation at the same time.
9. Overall, don't forget about the team spirit and don't do to others what you would not like to be done to you.

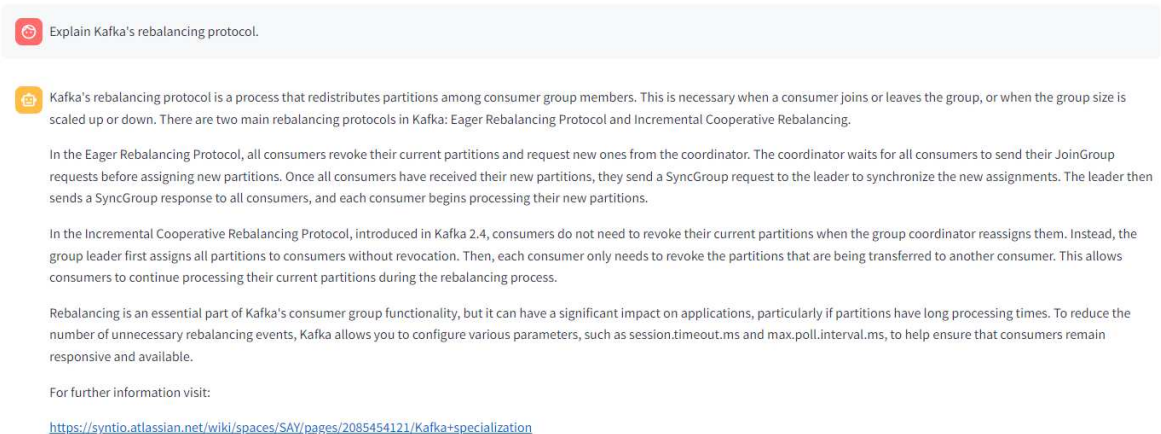
In summary, taking vacation days in Syntio requires careful planning, coordination with your team, and approval from your Team Lead. Make sure to communicate your absence to all relevant parties and ensure that your work is covered before you go.

For further information visit:

<https://syntio.atlassian.net/wiki/spaces/SAY/pages/2425323661/Vacation+and+absence+guidelines>

Figure 5.3: Question about company's vacation policy

Besides the company-specific information, the system was tested on "general knowledge" data. Even though the model can answer these questions without the provided context, answers with context are more relevant and can provide more important and specific information. In the example shown on 5.4 and 5.5, the system provides a correct answer without the context. However, the contextual answer provides more depth and mentions more specific information (such as the Kafka version and relevant configuration parameters). Since internal knowledge materials often include findings from conducted research or detailed documentation that may not be widely available, contextual answers can prioritize information that is considered important within the organization.



The screenshot shows a chat interface with a light blue header containing a question: "Explain Kafka's rebalancing protocol." Below the question, an answer is provided, starting with "Kafka's rebalancing protocol is a process that redistributes partitions among consumer group members. This is necessary when a consumer joins or leaves the group, or when the group size is scaled up or down. There are two main rebalancing protocols in Kafka: Eager Rebalancing Protocol and Incremental Cooperative Rebalancing." The answer then details the Eager Rebalancing Protocol and the Incremental Cooperative Rebalancing Protocol, and ends with a link to a wiki page for further information.

Explain Kafka's rebalancing protocol.

Kafka's rebalancing protocol is a process that redistributes partitions among consumer group members. This is necessary when a consumer joins or leaves the group, or when the group size is scaled up or down. There are two main rebalancing protocols in Kafka: Eager Rebalancing Protocol and Incremental Cooperative Rebalancing.

In the Eager Rebalancing Protocol, all consumers revoke their current partitions and request new ones from the coordinator. The coordinator waits for all consumers to send their JoinGroup requests before assigning new partitions. Once all consumers have received their new partitions, they send a SyncGroup request to the leader to synchronize the new assignments. The leader then sends a SyncGroup response to all consumers, and each consumer begins processing their new partitions.

In the Incremental Cooperative Rebalancing Protocol, introduced in Kafka 2.4, consumers do not need to revoke their current partitions when the group coordinator reassigns them. Instead, the group leader first assigns all partitions to consumers without revocation. Then, each consumer only needs to revoke the partitions that are being transferred to another consumer. This allows consumers to continue processing their current partitions during the rebalancing process.

Rebalancing is an essential part of Kafka's consumer group functionality, but it can have a significant impact on applications, particularly if partitions have long processing times. To reduce the number of unnecessary rebalancing events, Kafka allows you to configure various parameters, such as session.timeout.ms and max.poll.interval.ms, to help ensure that consumers remain responsive and available.

For further information visit:

<https://syntio.atlassian.net/wiki/spaces/SAY/pages/2085454121/Kafka+specialization>

Figure 5.4: Question about Kafka's rebalancing protocol with context

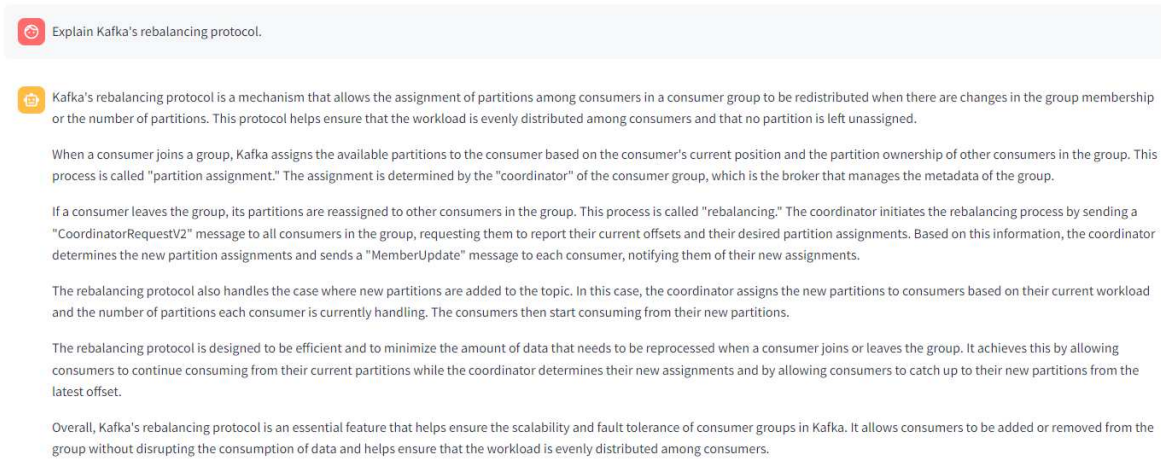


Figure 5.5: Question about Kafka's rebalancing protocol without context

Testing showed that the system sometimes fails to answer certain questions, even though the information is present in the knowledge base. In Figure 5.6, the question about the budget for buying headphones is not answered, with the explanation that the context does not provide the requested information. The likely issue here is the formatting of the benefits page, which uses a large table to present the data. Semantic parsing of the data in tabular format seems to be challenging for the system.



Figure 5.6: Question about the benefits budget

Another example is presented in Figure 5.7, which involves a question about the team lead for a specific Summer School team. The system was unable to provide the name, stating that the context does not mention it. Again, this information exists in the knowledge base, but the page with information about teams and their members is formatted in bullet points, showing that the sentence chunker used by the system might not be the most effective tool for parsing such formats.



Who was the team lead of team Music in summer school 2022?



The context provided does not mention the team lead of team Music in summer school 2022.

For further information visit:

<https://syntio.atlassian.net/wiki/spaces/SAY/pages/2093940753/1.7.+Agile+Development>

<https://syntio.atlassian.net/wiki/spaces/SAY/pages/2214854657/Mentor%27s+problems+and+tips>

<https://syntio.atlassian.net/wiki/spaces/SAY/pages/2382135297/Guidelines+-+2023>

Figure 5.7: Question about Summer School team leads

6 Discussion

6.1 Implications

The design and implementation of this system serve as a proof-of-concept for incorporating RAG-based LLM systems in business environments. The implementation of the RAG system in an internal knowledge base showed multiple benefits, including more relevant search results with references to original Confluence pages, and minimized risks of hallucinations. The system could improve search efficiency within large business knowledge bases; by combining LLM models with a retrieval mechanism, the RAG approach can offer more relevant search results faster than traditional key-word search methods. Additionally, every answer includes references to the original Confluence pages in the form of links, allowing users to verify the information and gain quick access to related content. In cases where the system cannot provide an answer, it still provides links to related pages that might contain the necessary details, enabling users to explore the topic further. The LLM used is configured to minimize the risk of hallucinations; when the system lacks sufficient context to provide a reliable answer, it avoids fabricating information and instead clearly states the limitations of the provided context.

6.2 Future Direction

One potential area for future improvement of the RAG system is the implementation of hybrid search (instead of just vector search). Hybrid search (most commonly) refers to the combination of traditional keyword-based search and modern semantic search. By combining keyword-based search which excels in exact keyword-matching scenarios but struggles with synonyms and contextual details, and vector search which offers semantic context awareness but may miss essential keywords and depend on the qual-

ity of generated embeddings, the hybrid search could offer “the best of both worlds”, at least in theory. Although the concept of hybrid search presents a promising approach to optimizing search and retrieval, its practical implementation faces challenges, particularly concerning compatibility and usability issues with existing vector databases. Qdrant, the vector database used, does not support hybrid search out-of-the-box. This limitation meant that integrating hybrid search would require manual implementation, significantly increasing the complexity of the system. Moreover, hybrid search is most beneficial in domains and scenarios where many rare keywords or specialized terms are present. For example, in the medical domain, numerous rare terms and specialized vocabulary are not typically found in general language models. In such contexts, hybrid search can ensure that these specific terms are accurately matched while still benefiting from the contextual understanding provided by semantic search. Given that the current implementation of the RAG system is not focused on such specialized domains, the need for hybrid search was less critical. However, a possible future direction is to implement hybrid search, either by using a different vector database that has native hybrid search or by implementing it manually.

Another possible improvement of the quality and relevance of the retrieved chunks could be facilitated by the application of *reranking methods*, which involves reordering retrieved results based on some criteria. The main reasoning for incorporating reranking methods is that by considering only the top k responses from the knowledge base retrieval, we might miss out on potential valuable context; since Confluence pages are chunked for embeddings, it could happen that the context is lost across the smaller parts, or the relevant information is contained in more than k chunks.

As previously mentioned, one significant benefit of the RAG method is its ability to avoid the issue of stale and outdated data, ensuring that the answers remain relevant and correct. Although the current implementation does not include automated retrieval of up-to-date information from Confluence, this is surely an important step for future development. Automating this process would ensure that the RAG system always has access to the latest information, without manually reloading the knowledge database.

To make the system more *user-friendly*, it could be improved with conversational memory capabilities. This would let the system remember the context from previous interactions, providing more coherent and relevant responses. This feature would be especially

helpful for users with long or complex queries, as it would allow the system to maintain a continuous and logical flow of information, enhancing user satisfaction and engagement. For a RAG system in a production environment, implementing multi-tenancy is a crucial step. Multi-tenancy enables a single instance of the RAG system to serve multiple users, each with its own isolated data and history.

6.3 Future of RAG

As LLMs continue to develop, they are becoming more powerful, with newer models having longer input contexts and improved capabilities. A recent breakthrough is the release of Google's Gemini 1.5, a multimodal LLM with an impressive context window of up to 1 million tokens in production and up to 10 million tokens in research. This means Gemini 1.5 can process huge amounts of data, such as one hour of video, eleven hours of audio, a codebase with over 30,000 lines of code, or texts containing over 700,000 words [36]. In evaluations, Gemini 1.5 has outperformed other RAG-powered LLMs in answering questions about large collections of text across various context sizes [37].

Despite these advancements, the growing context window size in LLMs doesn't mean the end of RAG. Even though it might look like LLMs with larger context windows could take over, there are still many challenges and limitations with these large context LLMs. This shows that RAG is still relevant and necessary. Firstly, even the largest models struggle to achieve sub-second response times when dealing with long contexts. The volume of data that these models must handle can slow down their response times, which is a critical issue in applications that require quick responses. Furthermore, generating high-quality answers within long contexts is very computationally expensive. For instance, retrieving 1 million tokens of data at a rate of \$0.0015 per 1,000 tokens can lead to substantial expenses, potentially amounting to \$1.50 for a single request. This cost factor can quickly escalate, making the deployment of large context LLMs economically unfeasible for many applications [38]. Additionally, just increasing the context size does not address the issue of using sensitive proprietary data as a knowledge base. As a result, RAG methods continue to play a crucial role in the development and deployment of AI systems. The future of RAG is closely connected to current advancements in LLMs. As LLMs evolve, RAG methods are also developing to leverage these improvements.

7 Conclusion

The integration of Large Language Models and the RAG method represents a novel approach to enhancing business systems. While the emergence of LLMs like GPT models, PaLM, and Llama has revolutionized the field of NLP with their capabilities, their limitations in handling dynamic and domain-specific queries have led to the exploration of more sophisticated approaches. The RAG method, by integrating external data sources at the point of inference, addresses these limitations effectively, offering a promising solution for enhancing business applications. This thesis has demonstrated that while LLMs possess great potential, their reliance on static, pre-trained data renders them less effective for tasks requiring up-to-date or highly specific information. This limitation is particularly notable in business environments, where the critical information often resides in internal documents that are not included in the public datasets used to train these models. The implementation of a RAG-based system within a business context has shown multiple benefits, including improved search efficiency and more relevant results. Traditional keyword-based searches often fall short in extracting precise information from large, complex knowledge bases. In contrast, the RAG approach leverages both the power of LLMs and the specificity of real-time data retrieval, significantly enhancing the quality of the information retrieved. Additionally, the inclusion of references to original sources within responses not only adds a layer of verifiability but also allows users to explore further, enabling a deeper understanding of the queried information. This thesis has demonstrated the benefits of this approach while outlining a roadmap for future research and improvements. This paves the way for even smarter and more effective use of LLM technology in business. As these systems keep evolving, they have the exciting potential to increase efficiency and insight, changing the way businesses handle and utilize their internal knowledge.

References

- [1] “Chatgpt continues to be one of the fastest-growing services ever,” <https://www.theverge.com/2023/11/6/23948386/chatgpt-active-user-count-openai-developer-conference>, [Online; Accessed: May 2024].
- [2] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. T. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” vol. 2020-December, 2020.
- [3] A. Mallen, A. Asai, V. Zhong, R. Das, D. Khashabi, and H. Hajishirzi, “When not to trust language models: Investigating effectiveness of parametric and non-parametric memories,” 2023.
- [4] G. Izacard, P. Lewis, M. Lomeli, L. Hosseini, F. Petroni, T. Schick, J. Dwivedi-Yu, A. Joulin, S. Riedel, and E. Grave, “Atlas: Few-shot learning with retrieval augmented language models,” 2022.
- [5] J. Menick, M. Trebacz, V. Mikulik, J. Aslanides, F. Song, M. Chadwick, M. Glaese, S. Young, L. Campbell-Gillingham, G. Irving, and N. McAleese, “Teaching language models to support answers with verified quotes,” 2022.
- [6] B. Bohnet, V. Q. Tran, P. Verga, R. Aharoni, D. Andor, L. B. Soares, M. Ciaramita, J. Eisenstein, K. Ganchev, J. Herzig, K. Hui, T. Kwiatkowski, J. Ma, J. Ni, L. S. Saralegui, T. Schuster, W. W. Cohen, M. Collins, D. Das, D. Metzler, S. Petrov, and K. Webster, “Attributed question answering: Evaluation and modeling for attributed large language models,” 2023.
- [7] “Retrieval augmented generation (rag) in azure ai search,” <https://learn.microsoft.com/en-us/azure/search/retrieval-augmented-generation-overview>, [Online; Ac-

cessed: December 2023].

- [8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020.
- [9] “What are vector embeddings,” <https://www.pinecone.io/learn/vector-embeddings/>, [Online; Accessed: May 2024].
- [10] “Meet ai’s multitool: Vector embeddings,” <https://cloud.google.com/blog/topics/developers-practitioners/meet-ais-multitool-vector-embeddings>, [Online; Accessed: June 2024].
- [11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023.
- [12] “What is a vector database?” <https://qdrant.tech/articles/what-is-a-vector-database/>, [Online; Accessed: May 2024].
- [13] J. J. Pan, J. Wang, and G. Li, “Survey of vector database management systems,” 2023.
- [14] “Picking a vector database: a comparison and guide for 2023,” <https://benchmark.vectorview.ai/vectordbs.html>, [Online; Accessed: May 2024].
- [15] “Vector db comparison,” <https://superlinked.com/vector-db-comparison>, [Online; Accessed: May 2024].
- [16] N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, “Mteb: Massive text embedding benchmark,” *arXiv preprint arXiv:2210.07316*, 2022. <https://doi.org/10.48550/ARXIV.2210.07316>
- [17] “Qdrant,” <https://qdrant.tech/>, [Online; Accessed: May 2024].
- [18] “Qdrant: Introduction,” <https://qdrant.tech/documentation/overview/>, [Online; Accessed: May 2024].

- [19] “Qdrant: Collections,” <https://qdrant.tech/documentation/concepts/collections/>, [Online; Accessed: May 2024].
- [20] A. F. Martin Aumeller, Erik Bernhardsson, “Ann benchmarks,” <https://ann-benchmarks.com/>, [Online; Accessed: May 2024].
- [21] “Indexing,” <https://qdrant.tech/documentation/concepts/indexing/>, [Online; Accessed: May 2024].
- [22] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, “Mistral 7b,” 2023.
- [23] “Sentence-transformers/all-mpnet-base-v2,” <https://huggingface.co/sentence-transformers/all-mpnet-base-v2>, [Online; Accessed: June 2024].
- [24] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, “Mpnet: Masked and permuted pre-training for language understanding,” 2020.
- [25] “Nltk documentation,” <https://www.nltk.org/api/nltk.tokenize.html>, [Online; Accessed: May 2024].
- [26] “Language processing pipelines,” <https://spacy.io/usage/processing-pipelines>, [Online; Accessed: May 2024].
- [27] “Recursively split by character,” https://python.langchain.com/v0.1/docs/modules/data_connection/document_transformers/recursive_text_splitter/, [Online; Accessed: May 2024].
- [28] “A guide to chunking strategies for retrieval augmented generation (rag),” <https://www.sagacify.com/news/a-guide-to-chunking-strategies-for-retrieval-augmented-generation-rag>, [Online; Accessed: May 2024].
- [29] “Building high-quality rag systems,” <https://implementconsultinggroup.com/article/building-high-quality-rag-systems>, [Online; Accessed: May 2024].

- [30] “Streamlit: Chat elements,” <https://docs.streamlit.io/develop/api-reference/chat>, [Online; Accessed: May 2024].
- [31] “Best practices for llm evaluation of rag applications,” <https://www.databricks.com/blog/LLM-auto-eval-best-practices-RAG>, [Online; Accessed: May 2024].
- [32] “Evaluating rag: Using llms to automate benchmarking of retrieval augmented generation systems,” <https://www.willowtreeapps.com/craft/evaluating-rag-using-llms-to-automate-benchmarking-of-retrieval-augmented-generation-systems>, [Online; Accessed: May 2024].
- [33] A. B. Sai, A. K. Mohankumar, and M. M. Khapra, “A survey of evaluation metrics used for nlg systems,” 2020.
- [34] P. Wang, L. Li, L. Chen, Z. Cai, D. Zhu, B. Lin, Y. Cao, Q. Liu, T. Liu, and Z. Sui, “Large language models are not fair evaluators,” 2023.
- [35] “[r-240] (docs): Document how to evaluating with a locally hosted llm to help choose the ones that work best,” <https://github.com/explodinggradients/ragas/issues/859>, [Online; Accessed: May 2024].
- [36] “Our next-generation model: Gemini 1.5,” <https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#build-experiment>, [Online; Accessed: May 2024].
- [37] G. T. et al, “Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context,” 2024.
- [38] “Will retrieval augmented generation (rag) be killed by long-context llms?” <https://zilliz.com/blog/will-retrieval-augmented-generation-RAG-be-killed-by-long-context-LLMs>, [Online; Accessed: May 2024].

Abstract

Transformation of Business Systems using Large Language Models and the RAG Method

Dora Horvat

This thesis presents the design and implementation of a Retrieval-Augmented Generation (RAG) based Large Language Model (LLM) system. By integrating retrieval mechanisms with generative capabilities, an innovative concept in the world of natural language processing and artificial intelligence is explored and tailored for business systems. The proposed solution aims to bridge the gap in business document search; knowledge sharing and collaboration tools mainly rely on keyword-based search, without the ability to semantically process user queries and understand the context of the questions. The RAG-based system employs semantic search methods to retrieve relevant documents and augment the generative model's responses, enhancing the accuracy and relevance of information provided to users. The implemented system showcases the potential of RAG systems in transforming business processes while also providing a framework for future research and development.

Keywords: Retrieval-augmented generation (RAG); Large language models (LLMs); Semantic search; Vector databases; Business document search

Sažetak

Transformacija poslovnih sustava pomoću velikih jezičnih modela i metodi RAG

Dora Horvat

U ovom radu predložen je dizajn i prikazana je implementacija sustava koji proširuje sposobnosti velikih jezičnih modela metodom generiranja potpomognutog pretraživanjem (RAG). Integracijom mehanizma dohvaćanja s generativnim sposobnostima jezičnih modela istražuje se inovativan koncept u svijetu obrade prirodnog jezika i umjetne inteligencije, te se prilagođava poslovnim sustavima. Predloženo rješenje nastoji premostiti postojeći jaz u pretraživanju poslovnih dokumenata; alati za dijeljenje znanja i suradnju uglavnom se oslanjaju na pretraživanje temeljeno na ključnim riječima, bez mogućnosti semantičke obrade korisničkih upita i razumijevanja konteksta pitanja. Sustav temeljen na RAG metodi koristi sematičke metode pretraživanja kako bi pronašao relevantne dokumente i poboljšao odgovore generativnog modela, povećavajući točnost i relevantnost informacija koje se pružaju korisnicima. Implementirani sustav prikazuje potencijal RAG sustava u transformaciji poslovnih procesa, istovremeno pružajući okvir za buduća istraživanja i razvoj.

Ključne riječi: Generiranje potpomognuto pretraživanjem (RAG); Veliki jezični modeli (LLM); Semantičko pretraživanje; Vektorske baze podataka; Pretraživanje poslovnih dokumenata