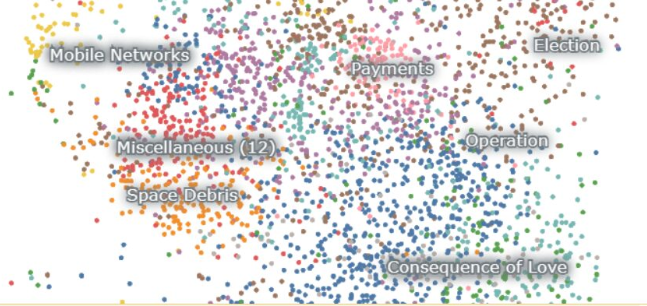
**Embeddings**

embeddings are a useful machine learning concept involved in representing data as points in an n-dimensional space.

 Embeddings are vectors or arrays of numbers that represent the meaning be it sentences from a document, images or audio, in a higher dimensional space.

These embeddings exist in a space with many dimensions, where each dimension signifies a feature or characteristic that the model has learnt about from the data.

[Embedding Demonstration](https://atlas.nomic.ai/map/53d8f32e-df36-4394-b30c-6aa4c51968fa/37fc4954-9d86-4331-b204-925709e92883)



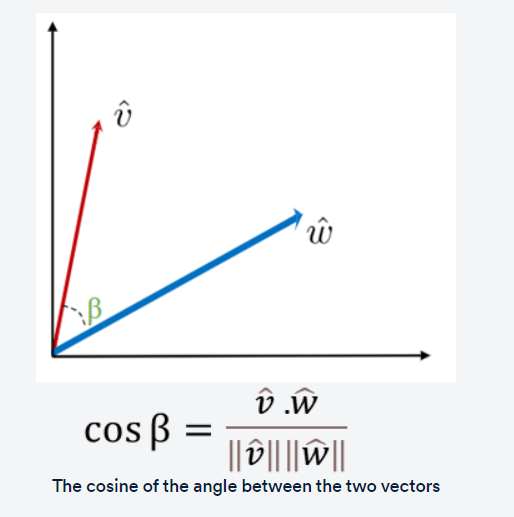
Embeddings are how a model grasps the meaning and connections within language as well as how it evaluates and distinguishes between various language components. They act as a link connecting the discrete and continuous aspects as well as the symbolic and numeric elements of language for the models.

**Similarity Between two vectors**

embeddings are not just abstract representations of words - they capture the meaning and relationships within language. This is achieved through extensive training on vast amounts of text data.

During this training, models learn to position words or phrases in a vector space such that similar words are closer to each other, whereas dissimilar ones are further apart. Once the words and pieces of text have been represented as vectors in a high-dimensional space, their similarity or dissimilarity can be measured using various distance metrics.

Cosine similarity is a popular method for measuring similarity between vectors. For two vectors - u, w, the cosine of the angle between the two vectors is given using the following formula shown in the image below.



If the vectors point in the same direction (i.e., they are similar), the cosine similarity is close to 1. If the vectors are orthogonal (90° apart, indicating dissimilarity), the cosine similarity is 0, and if they point in opposite directions (indicating strong dissimilarity), the cosine similarity is −1.

**Vectors:**

Vector representations for words are a powerful tool in natural language processing. It allows us to capture the meaning of words in a way that can be used by machine learning algorithms.

**Word embeddings**

Vector representations for words, also known as word embeddings, are a way of representing words as vectors in a high-dimensional space.

Each dimension of the vector represents a different feature of the word, such as its meaning, context or syntactic role. By representing words as vectors, we can perform mathematical operations on them, such as addition and subtraction, to capture the relationships between words and to perform various natural language processing tasks.

vector representations of words are useful for capturing the context of the word in a sentence. These are particularly useful when words have multiple meanings. These are referred to as word embeddings and are useful for representing words and capturing semantics.

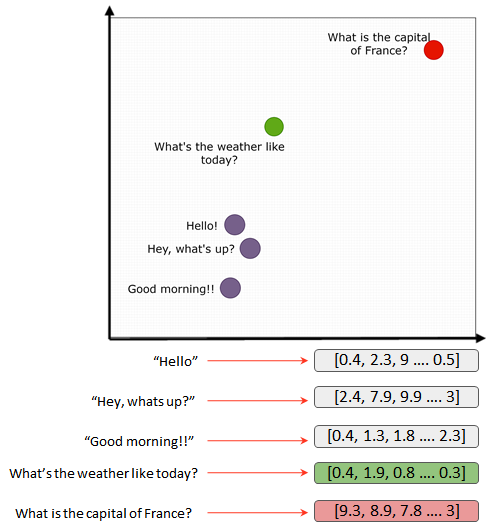
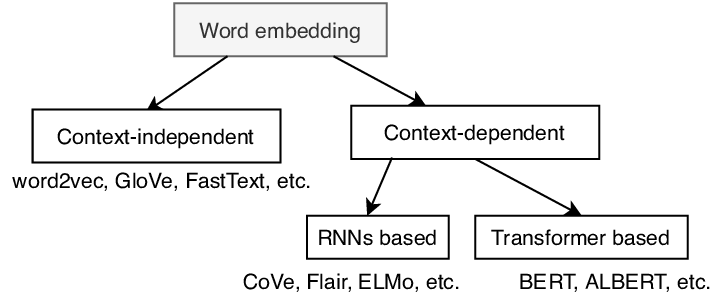
Word embeddings are a way of representing words as vectors in a high-dimensional space, where each dimension corresponds to a feature/set of features representing the word. These features can be anything from the frequency of the word in a corpus to its semantic meaning. The key idea underlying the principle of word embeddings is that words with similar meanings and contexts tend to be grouped close together in a high-dimensional space.

They are created using algorithms such as Word2vec and GloVe and have many applications in information retrieval, language modelling and other areas of NLP.

By representing words as vectors, we can enable computers to understand the meaning and context of texts, which, in turn, has led to many practical applications, such as text classification, sentiment analysis and machine translation.

**Sentence embeddings**

Sentence embeddings generally perform better on individual sentences than their counterparts, i.e. word embeddings that only focus on individual words and tokens.

**Semantic Search**

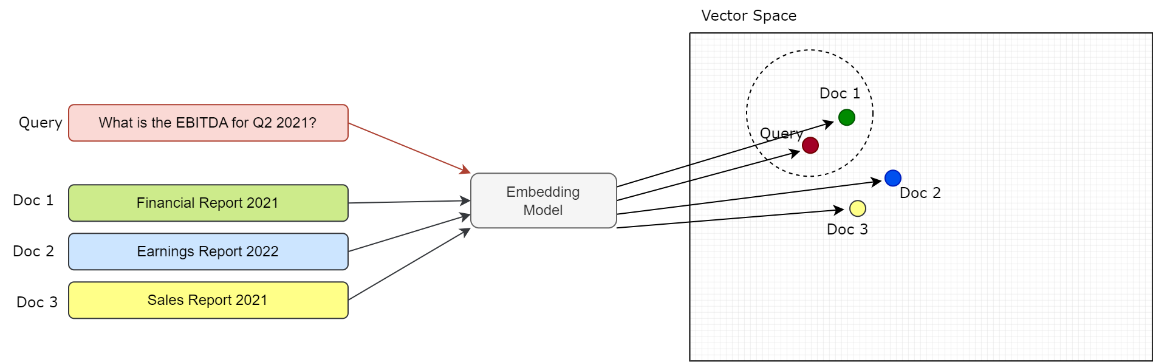
Search, or the more technical term ‘information retrieval’, is the process of retrieving relevant information from a large collection of data.

Semantic search can be considered to be a subset of information retrieval that seeks to improve search accuracy by understanding the user's intent and the contextual meaning of terms by applying the principles of embedding space.

Semantic search uses vector search and machine learning techniques to return results that aim to match a user’s query even when there are no word matches

Semantic search focuses on the semantic understanding of the query and the documents by generating and comparing vector embeddings. As word embeddings that can be used to represent words as vectors in a high-dimensional space. These embeddings can be used to capture the **semantic meaning** of words and measure the similarity between words or documents.

In semantic search ( sometimes also known as dense retrieval), the objective is to extract the document that matches closely with the user’s query. Semantic search systems go beyond simple keyword matching and take into consideration the context, semantics and conceptual relationships between words to match a user query with the corresponding content. As a result, a semantic search system can understand the intent and meaning behind a user's query and match it with relevant documents even if the query and the documents use different words or phrasing. After representing the pieces of text or documents in a high-dimensional space, the vector embeddings of the query and the documents can be compared using a distance metric such as cosine similarity. The search problem is now converted to a nearest neighbour method to find the phrase that closely matches with the vector embeddings of the query phrase and vice versa, as illustrated in the diagram below.

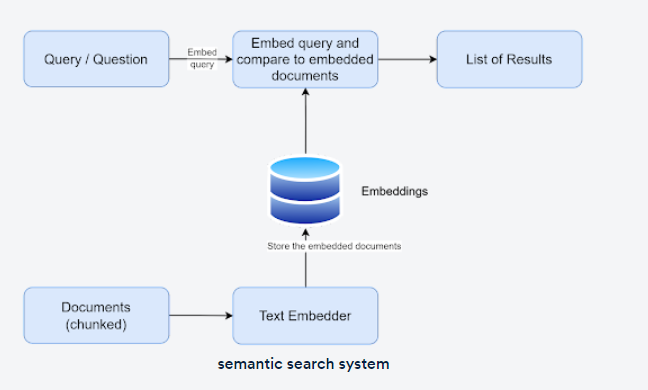


**Semantic search pipeline**

semantic search system, we will use embedding models to extract the embedding of multiple documents and map them to the appropriate locations in the embedding space.

The query is then mapped using the same embedding model and compared to get the closest matching document.

The general flowchart for the semantic search system is illustrated below.

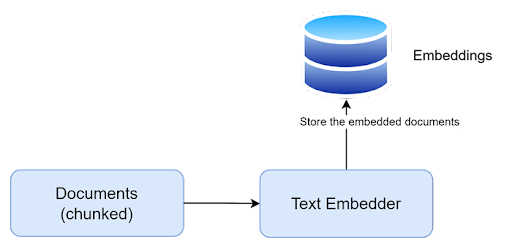


The two stages of a semantic search pipeline are:

1. Encoding pipeline
2. Search/ decoding pipeline

The encoding pipeline deals with ingesting the documents for search, generating the vector embeddings and storing them for later retrieval. The pipeline consists of the following steps as shown in the image below:

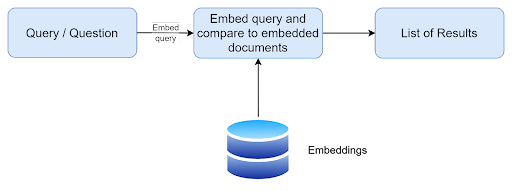
1. Collect documents for embedding
2. Create text embeddings to encode semantic information
3. Store embeddings in a database for later retrieval upon receiving a query



[METB leaderboard](https://huggingface.co/spaces/mteb/leaderboard) is a platform provided by Hugging Face that showcases the performance of various text embedding models on a range of tasks and datasets. The leaderboard offers a holistic view of the best text embedding models available, allowing researchers and practitioners to compare and analyse their performance.

Once the embeddings have been generated, you can store them locally for later retrieval through the search or **decoding pipeline**. The search or decoding pipeline compares the vector embeddings of the query against the vector embeddings contained within the document. It generally consists of the following steps:

1. Retrieve the user’s query.
2. Compare the embeddings of the query and the document embeddings generated from the encoding pipeline.
3. Retrieve the relevant candidate documents using an appropriate distance metric.
4. Return the final search results.



first element in the encoding pipeline - the text embedder. The text embedding model is at the heart of any semantic search system. The embedding model takes in a word, phrase, sentence or multiple sentences from a document and then converts them into a vector representation. This vector representation is unique and captures the inherent contextual meaning.

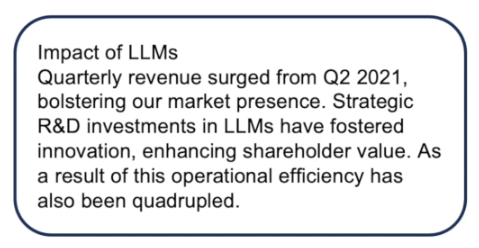
There are open-source sentence transformers made available through the Hugging Face library as well as closed-source offerings such as Cohere or OpenAI that can be accessed via APIs. These embedding models vary in terms of their architectures, training data and number of dimensions used for representing words as vectors.

Some embedding models have unique properties, owing to which they are easier to work with. For instance, the magnitude of vectors in OpenAI embeddings is normalised to a length of 1, which leads to the cosine similarity metric being identical to the dot product.

Once a suitable text embedding model has been selected, the next step is to process the documents/ collection of documents and integrate them into the semantic search pipeline.

there are caveats to process documents while creating the embeddings for a semantic search application.

the example of a chunk of text from a document.



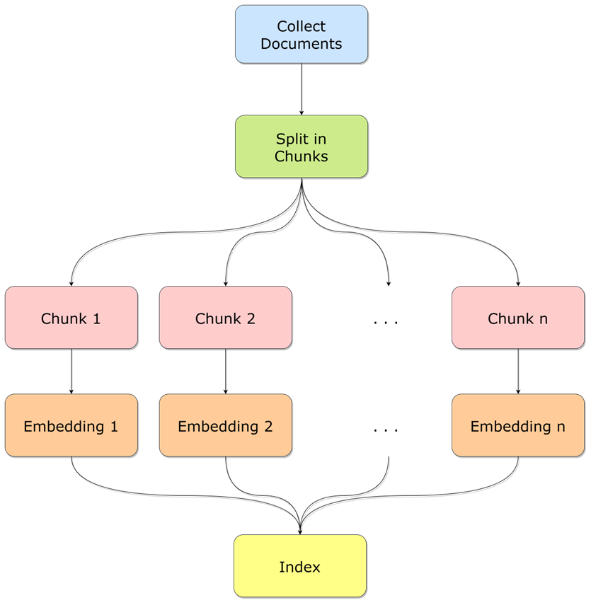
 the individual sentence in this chunk has varied contexts. Hence a proper document chunking strategy is very crucial in any semantic search system.

**DOCUMENT INGESTION**

**Document Chunking**

Chunking strategy is crucial to determine the effectiveness of a semantic search application. A chunk is a piece of text that is representative of the information contained within that section of the text.

A chunking strategy involves dividing a large document into smaller, more manageable chunks for embedding, as illustrated in the diagram below.

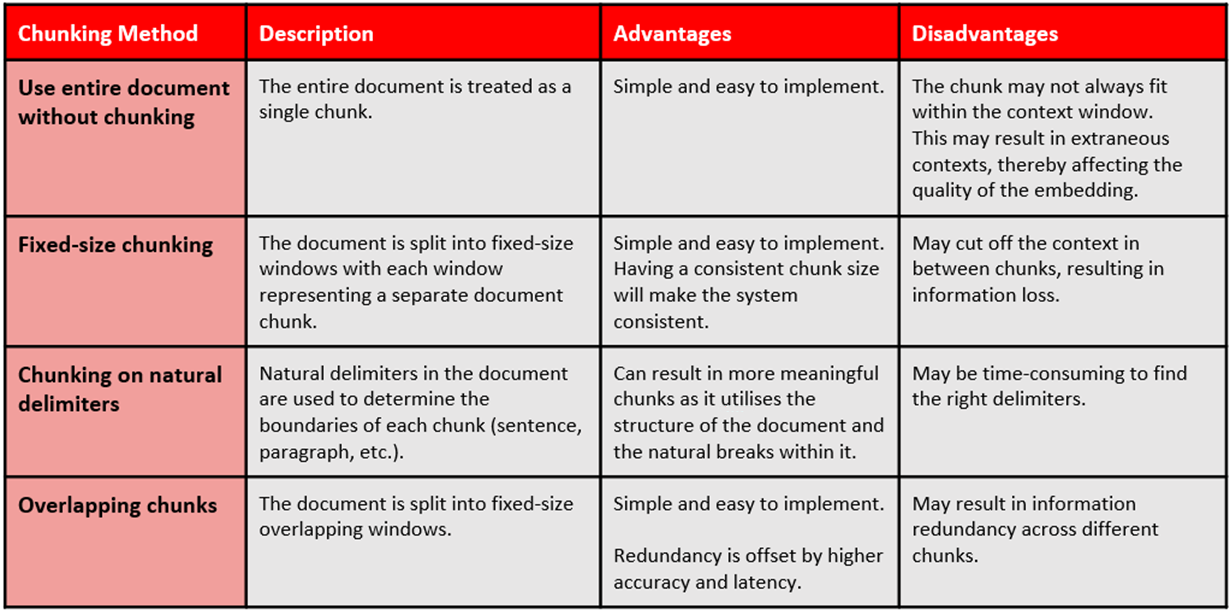


There can be various factors that influence the choice of the chunking strategy:

* The embedding model and its optimal chunk size
* Nature of the context
* Downstream task

Most embedding models have a context window limit. A text embedding model represents words as vectors by extracting the context from the surrounding words or text. Oftentimes, the context window size of the text embedding models needs to be considered before any document chunking strategy is taken into consideration. The size of the context window largely influences the resulting embeddings - a larger window size can result in the model representing the broader, topical meaning of the word, whereas smaller window sizes will tend to give results that reflect more focused information about the target word. The choice of context is a crucial factor that directly affects the resulting vector representations.

* 4 chunking strategies: the fixed window strategy, the max token window strategy, natural delimiter based strategy and Overlapping window chunking



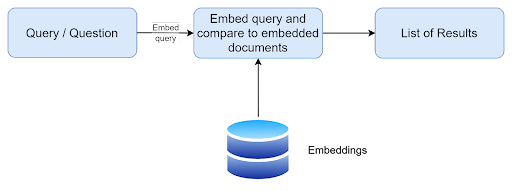
**Generating Embeddings**

the next step of the encoding pipeline - generating the embeddings for the chunked data.(refer page 1 to know about embeddings)

generate the vector embeddings for the chunked data using sentence transfomers' embedding models. Once the embeddings were generated, you stored them in a pandas dataframe for easy retrieval.

With this, the encoding pipeline for your semantic search application is complete.

**DECODING / SEARCH PIPELINE**



the query vector and the vector embeddings for various chunking strategies can be compared using a simple cosine similarity to retrieve semantically similar results.

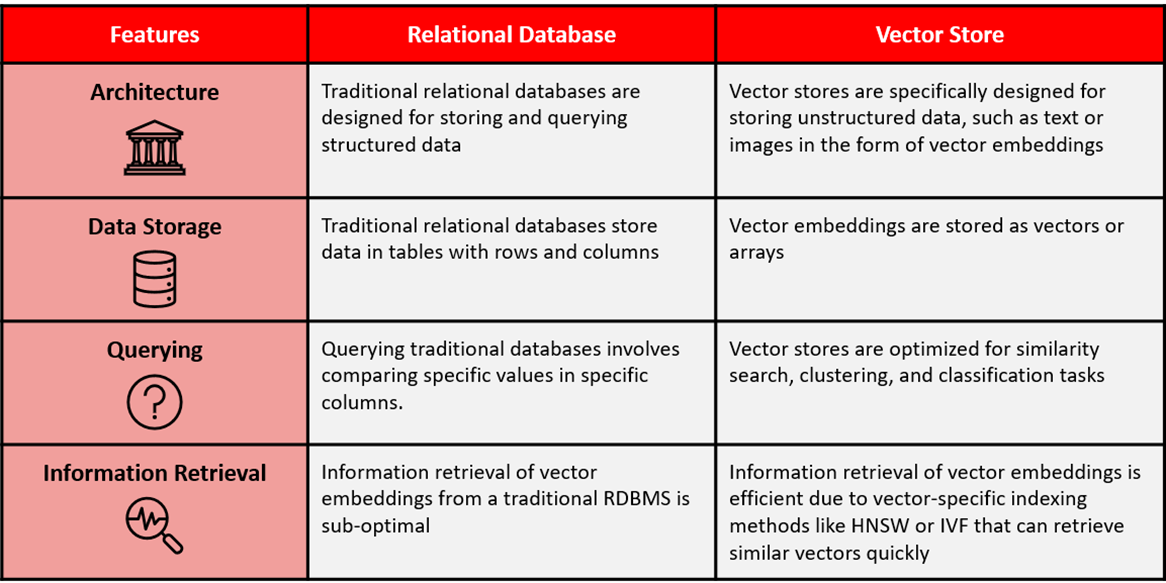
semantic search by comparing a query against the chunked and embedded documents.

The time complexity in such a scenario is the linear function of the number of entries **(i.e., O(N)).**

documents that contain millions of entries and span even more chunks and embeddings, such a solution leads to latency issues when search results are usually expected in the millisecond range. In such situations**, vector store**s can be considered for building a semantic search application

**Vector Database**

a vector store is a data storage system that is specially designed to store, index and retrieve high-dimensional vectors quickly. They can store and retrieve vectors much faster than traditional relational databases (RDBMS) and local storage options such as dataframes or flat files (.csv, .xlsx etc.). The image below illustrates some of the key differences between relational databases and vector stores.



They offer fast and accurate similarity search and data retrieval based on their vector distance or other similarity metrics. Vector stores consist of various components that work together to provide efficient storage, indexing and querying capabilities for high-dimensional vectors. Some of its key features include data management, metadata storage and filtering, and approximate nearest-neighbour (ANN) search algorithms.

* **Data management**: Vector databases offer features for easy data storage, insertion, deletion and updating, making it convenient to manage and maintain vector data.
* **Metadata storage and filtering**: These databases can store metadata associated with each vector entry, allowing users to query the database using additional metadata filters for more precise queries.
* **Approximate nearest-neighbour (ANN) search algorithms**: Vector databases use a combination of algorithms to optimise similarity search, such as hashing, quantisation or graph-based search.

The main advantages of using vector stores for storing and querying high-dimensional vectors are as follows:

* **Fast and accurate similarity search**: Vector stores excel at finding the most similar or relevant data based on the underlying semantic or contextual meaning of various texts which enables efficient retrieval of information. These can return query results faster than the traditional methods of search, such as keyword-based search or k-Nearest Neighbour-based searching methods.
* **Flexibility**: Vector stores can be used with various types of high-dimensional vectors, ranging from tens to thousands of dimensions, depending on the complexity and granularity of the data.

The popularity of vector stores is augmented by the availability of indexing strategies that can retrieve embeddings faster than traditional lookup-based approaches. Indexing in vector stores involves breaking down a document or website into smaller segments and converting these segments into vectors that can be stored in a vector database.

**Indexing Strategy**

indexing strategies to efficiently query vectors by computing the proximity of a query to the vector embeddings. The indexing algorithms used in vector databases vary depending on the specific application. Recently, however, approximate nearest-neighbours (ANN) methods such as product quantisation, Hierarchical Navigable Small World (HNSW) and Locative Sensitive Hashing(LSH) have garnered significant attention from developers and researchers alike. As the name suggests, ANN methods involve an approximation of the usual nearest-neighbour methods. You might already be familiar with some of the common methods of the nearest-neighbour algorithm called the k-nearest neighbour (kNN) method.

As discussed already, semantic search involves comparing the vector representations of a query and document by generating the vector embeddings and comparing the embeddings using a distance metric such as cosine similarity. Exact nearest neighbours, such as the kNN algorithm, can often help narrow down the retrieval process and produce accurate search results. But this accuracy comes at the expense of increased retrieval time. On the other hand, ANN methods sacrifice accuracy for speed.

Now, as mentioned in the previous session, such an exact search often results in an O(N) time complexity; however, ANN techniques result in a sub-linear time complexity O(log(N)). This is achieved with the help of special indexing techniques that make retrieval faster compared to traditional lookup-based methods. Indexing is like sorting a guest list by a certain characteristic, such as the first letter of their names or their closeness to you, so you can find your friends faster. Searching in vector stores involves querying the vector database to retrieve the most similar or relevant data based on their vector distance or similarity. The vector store compares the indexed query vector to the indexed vectors in a data set to find the nearest neighbours by applying a similarity metric of the indexed vectors.

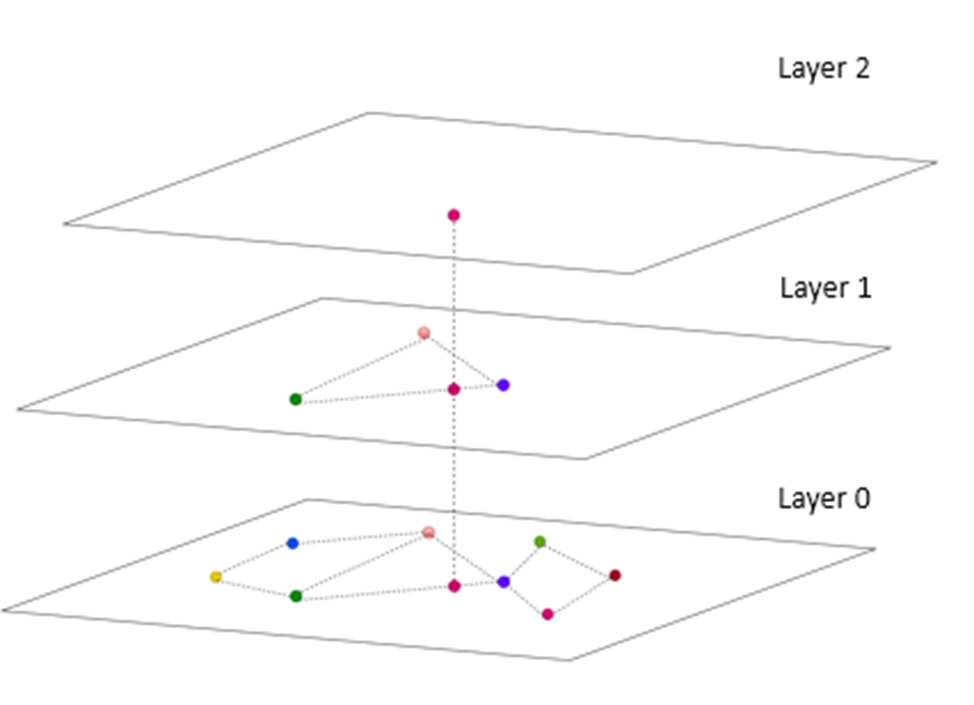
In summary, indexing is the process of organising vectors in a way that allows for efficient similarity search, while searching is the process of querying a vector database to retrieve the most similar or relevant data based on their vector distance or similarity. Some of the common approximate nearest-neighbour algorithms are as follows:

* Tree-based algorithms such as [ANNOY](https://github.com/spotify/annoy), which was created by Spotify
* Graph-based algorithms such as the Hierarchical Navigable Small World (HNSW) algorithm; popular C++ implementation of this algorithm available [here](https://github.com/nmslib/hnswlib)
* Cluster-based algorithms such as the [Facebook AI Similarity Search (FAISS)](https://github.com/facebookresearch/faiss) and [Product Quantisation](https://www.pinecone.io/learn/series/faiss/product-quantization/#:~:text=Product%20quantization%20(PQ)%20is%20a,x%20faster%20in%20our%20tests.)(PQ)
* Hash-based algorithms such as [Locality Sensitive Hashing](https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/) (LSH).

Each algorithm mentioned above finds applications in various use cases and comes with its own advantages and disadvantages. the popular algorithm HNSW, which is a popular method of conducting approximate nearest-neighbour searches.

**Hierarchical Navigable Small World (HNSW)**

The Hierarchical Navigable Small World (HNSW) algorithm is a popular graph-based method that combines the principles of Navigable Small World and proximity graphs. It is a fully graph-based solution that constructs a multi-layered graph with fewer connections in the top layers and more dense regions in the bottom layers as shown in the image below. The search starts from the highest layer and moves one level below every time the local nearest neighbour is found greedily among the layer nodes. Ultimately, the nearest neighbour found in the lowest layer is the answer to the query. Nodes in HNSW are inserted sequentially one by one, and every node is randomly assigned an integer indicating the maximum layer at which the node can be present in the graph.



The HNSW greedy search algorithm is sublinear, which means it has a complexity close to log(N), where N is the number of vectors in the graph. This makes it an efficient algorithm for approximate nearest-neighbour search. HNSW is used in various vector databases and libraries, including Pinecone, Faiss and ChromaDB.

**Vector Library**

Vector Libraries store vector vector embeddings in in-memory indexes, in order to perform similarity search

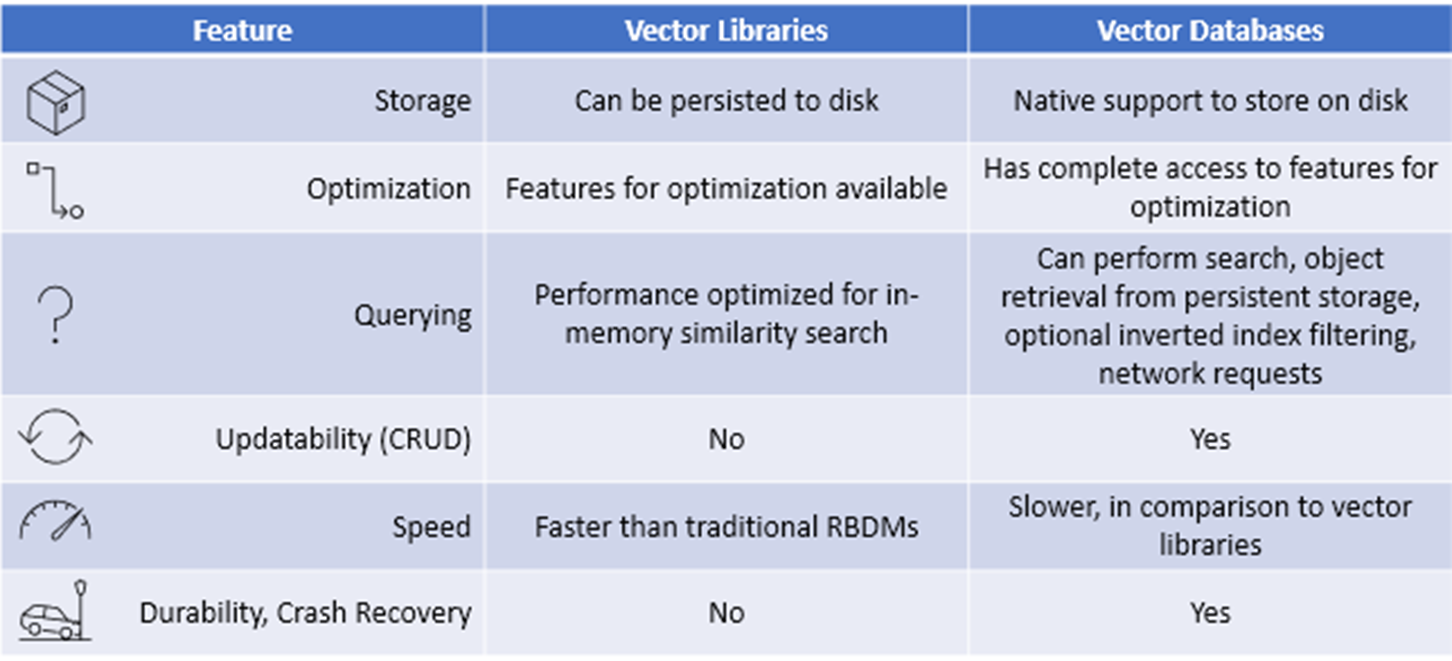
Vector stores come in multiple flavours; two common categories are **vector libraries and vector databases.**

Vector libraries and vector databases can be used to efficiently perform nearest-neighbour searches to retrieve similar pieces of text based on their semantic meaning.

Vector libraries (also referred to as vector indices or vector search libraries) are popular for quick prototyping purposes and when the data size is considerably small. These libraries do not support the usual create, read, update and delete (CRUD) support that traditional relational databases and vector databases offer; hence, they are not suitable for building scalable applications.

They, however, offer native support to store the vector embeddings to the local disk by persisting it from memory to the local disk.

**Vector databases** are optimised for storage and the retrieval of vector embeddings with the additional capabilities to store and update the vector embeddings, as they support CRUD operations natively. This makes them a great choice for applications that require low-latency search, such as recommendation systems, search engines and chatbots. Vector databases are typically more focussed on enterprise-level production deployments as opposed to vector libraries that are used for quick prototyping.

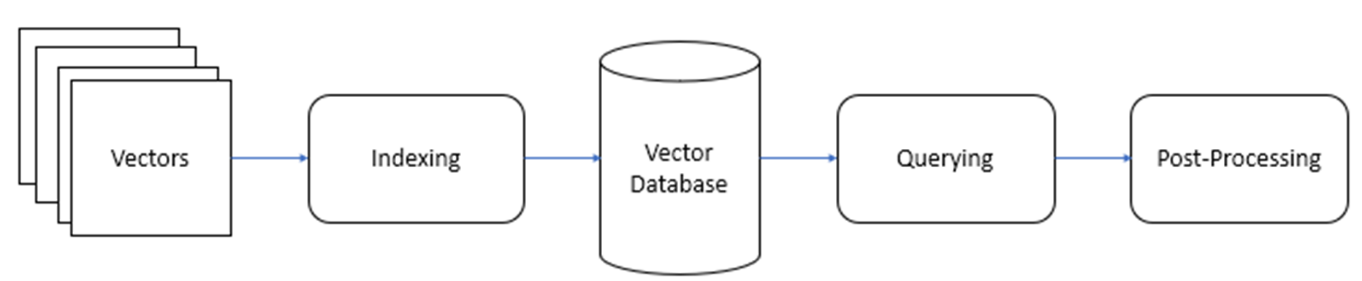


Additionally, vector databases solve several major limitations of vector libraries. These include:

* **Metadata storage and filtering**: Vector databases can store metadata associated with each vector entry. Users can then query a database using additional metadata filters for finer-grained queries. Vector libraries store only vector embeddings and not the associated objects they were generated from. When you run a query, a vector library will respond with the relevant vectors and object IDs. This is limiting since the actual information is stored in the object and not the ID. To solve this problem, you would have to store the objects in a secondary storage. You could then use the returned IDs from the query and match them to the objects to understand the results.
* **Scalability**: Vector databases are designed to scale with growing data volumes and user demands, providing better support for distributed and parallel processing. Standalone vector indices may require custom solutions to achieve similar levels of scalability (such as deploying and managing them on Kubernetes clusters or other similar systems).
* **Real-time updates**: Vector databases often support real-time data updates, allowing for dynamic changes to the data, whereas standalone vector indexes may require a full re-indexing process to incorporate new data, which can be time-consuming and computationally expensive.
* **Backups and collections**: Vector databases handle the routine operation of backing up all the data stored in a database. Pinecone also allows users to choose specific indexes that can be backed up in the form of ‘collections’, which store the data in that index for later use.
* **Ecosystem integration**: Vector databases can more easily integrate with other components of a data processing ecosystem, such as ETL pipelines (like Spark), analytics tools (like Tableau and Segment) and visualisation platforms (like Grafana), streamlining the data management workflow. It also enables easy integration with other AI-related tools such as LangChain, LlamaIndex and ChatGPT’s plugins.
* **Data security and access control**: Vector databases typically offer built-in data security features and access control mechanisms to protect sensitive information, which may not be available in standalone vector index solutions.

Here are the steps in the typical process of storing vector embeddings in a vector database:

1. **Generate vector embeddings**: In this step,  you generate the vector embeddings for the documents.
2. **Perform indexing**: The vector database indexes vectors using an algorithm such as PQ, LSH or HNSW (more on these below). This step maps the vectors to a data structure that will enable faster searching.
3. **Store indices and embedding vectors**: In this step, you store the embedding vectors in the local storage or cache of the vector store and generate the indices for the embeddings.
4. **Querying**: The vector database compares the indexed query vector to the indexed vectors to find the nearest neighbours (applying a similarity metric used by that index)
5. **Perform post-processing**: In this stage, common data management techniques such as updation and deletion operations are performed on the vector embeddings. In some cases, the vector database retrieves the final nearest neighbours from the data set and post-processes them to return the final results. This step can include re-ranking the nearest neighbours using a different similarity measure.



**Types of Vector DB**

**Chroma and Pinecone** are vector databases that are designed for small to medium-sized data sets.

**ChromaDB** is fully open-source, whereas Pinecone has a free tier and pricing plans that provide additional features and increased scalability. There are several open-source alternatives to Pinecone and ChromaDB that can be used to build a vector database for LLM (Large Language Model)-based embeddings.

One such alternative is **Pgvector**, a PostgreSQL extension that supports vector data types and provides fast vector operations.

Another option is **Weaviate**, a cloud-native, open-source vector database that is designed for machine learning applications. Weaviate supports semantic search and can be integrated with other machine learning tools such as TensorFlow and PyTorch.

**ANNOY** is an open-source library for approximate nearest-neighbour search that is optimised for large-scale data sets. It can be used to build a custom vector database tailored to specific use cases.

[**CHROMADB**](https://docs.trychroma.com/)

General usage of Chroma is as follows:

• Create collection to store embeddings: In this step, we create a collection, which is the equivalent of a table in a relational database. In this process, we indicate the model that Chroma should use to convert the texts into embeddings.

• Collection Management (Updating and Removing Data): In this stage, the text that has to be saved, along with whatever metadata you want for filtering the text is sent to Chroma. When Chroma receives the text, it will take care of converting it to embedding.

• Query: By sending a query or query embedding to Chroma, we will receive the most similar n documents. In addition, we can filter the query based on metadata so that it is only executed on the documents that meet a series of criteria.

* **Instantiate the Chroma client**: This method is used to instantiate the Chroma client by first initiating the Chroma package as shown in the code block below. Chroma can be configured to use an in-memory database or an on-disk database, which is useful for larger data that does not fit in the memory.

**import** **chromadb**

chroma\_client = chromadb.Client()

* **Adding to collection**: Once a client is created, a Chroma collection must be created to store the vector embeddings. A collection can be created using the ‘create\_collection’ method as shown in the following image.

collection.add(

embeddings = <embeddings>,

documents = <documents>,

metadata = <metadata>,

ids = <id>

)

* **Querying the collection**: Once the vector embeddings are stored in the collection, it can be queried using the ‘query’ method as shown in the image below. The query function can be used to search for similar documents based on a given query, which can be in the form of natural language or a specific embedding. When using the query function, you can specify which data you want returned, such as embeddings, documents, metadata and distances. By default, Chroma will return the documents, metadata and distances of the results while excluding the embeddings for performance reasons.

results = collection.query(

query = ['This is a query document'],

n\_results = **2**,

)

* **Collection management**: Chroma supports updation and deletion operations on the collection as shown in the image below. The ‘update’ method updates the vector embeddings by taking a dictionary with the new values for the item as an argument and the ID of the item to be updated. The delete method takes the ID of the item to delete as an argument, which will delete the embeddings, documents and metadata associated with the item.

collection.update(

ids = <id>,

documents = <document>,

metadata = <metadata>

)

collection.delete(

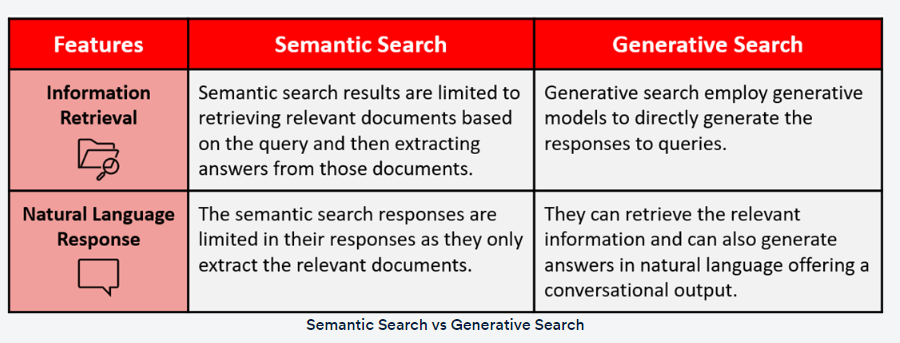
ids = <id>,

)

**Semantic search and Generative search**

Distinction between semantic search and generative search is that semantic search is primarily focused on retrieving relevant information, whereas generative search is focused on generating new content. However, the two technologies can be used together to improve the performance of a variety of tasks, such as question answering, summarisation, and machine translation.

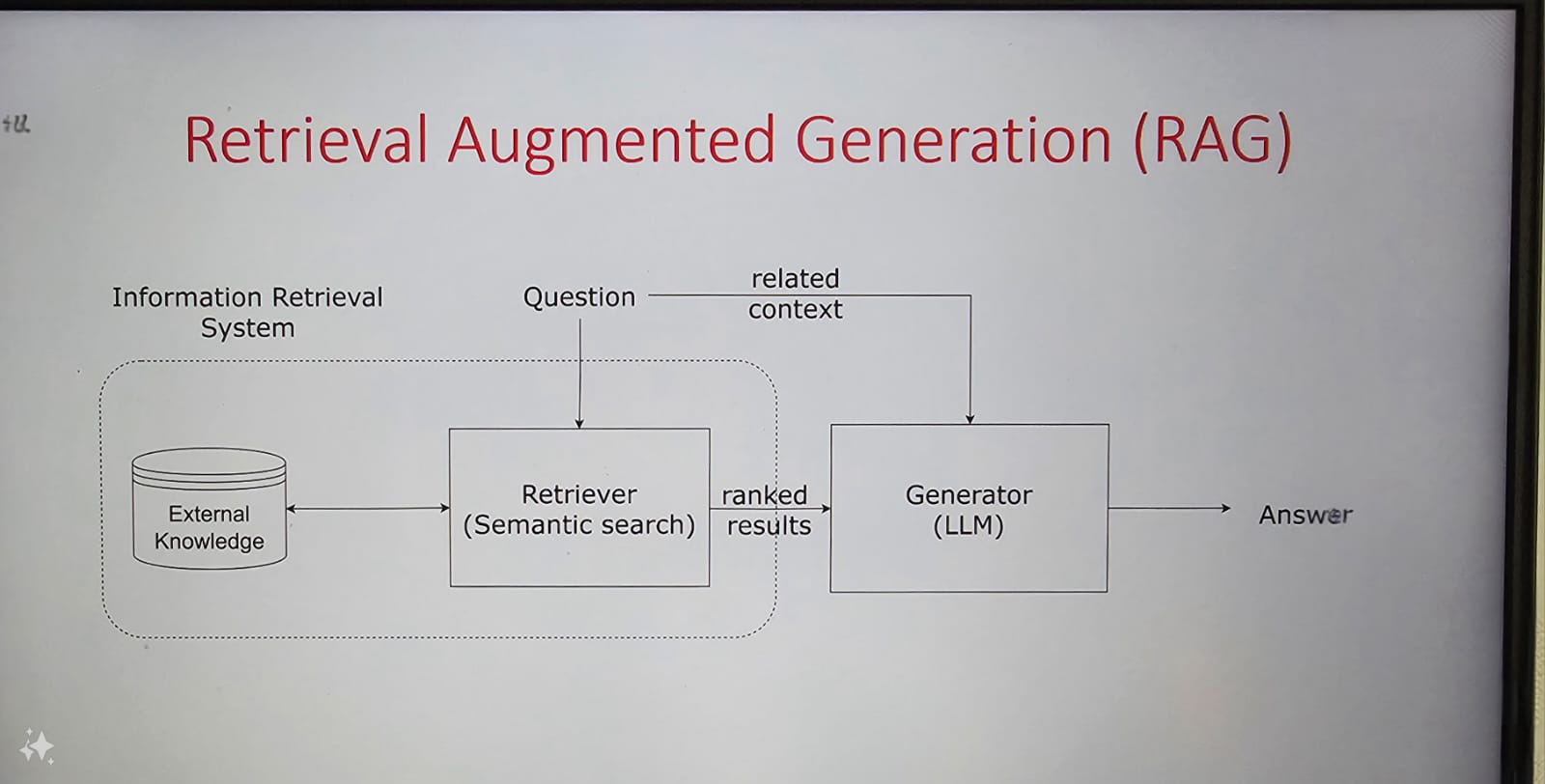
For example, a question-answering system can use semantic search to retrieve relevant documents from a knowledge base and then use generative search to generate a comprehensive and informative answer to the user's question



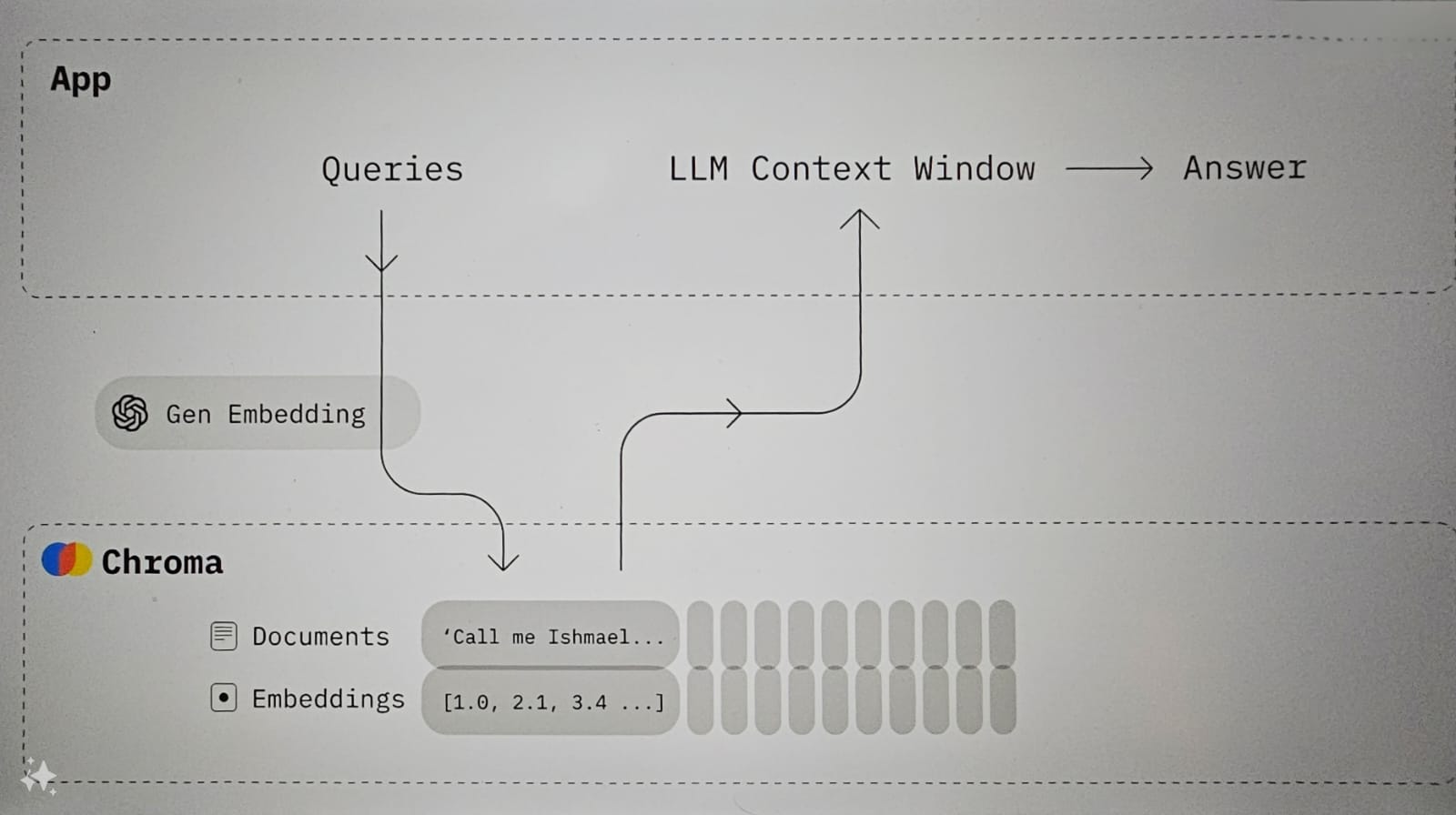
**Retrieval Augmented Generation (RAG)**

Retrieval augmented generation (RAG) is a special type of generative search that combines the strengths of semantic search and large language models to generate more accurate responses to user queries. This is a new search paradigm that combines the strengths of both retrieval-based models and generative foundation models to enhance the quality and relevance of the generated text. RAG retrieves relevant information from an external knowledge base to supplement the LLM's internal representation of information, which allows for fine-tuning and adjustments to the LLM's internal knowledge, making it more accurate and up-to-date. RAG has several applications, including question-answering systems, chatbots, and industry-specific LLMs. RAG can reduce hallucinations and repetition while improving specificity and factual grounding compared with conversation without retrieval. RAG can also provide more contextually appropriate answers to prompts as well as base those answers on the latest data.

**Generic/Basic Structure of RAG**

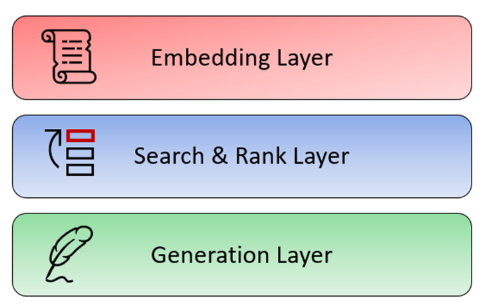
****

**Working of RAG**

****

**HELPMATEAI**

**Modules**



**Embedding Layer**

You are already familiar with the embedding layer, as it was covered in the previous sessions on semantic search. The embedding layer is typically the first layer of a RAG model, and it typically contains an embedding model that is trained on a massive data set of text and code. This data set is used to learn the relationships between words and phrases and to create embeddings that represent these relationships. The embedding layer is an important part of RAG models because it allows your system to understand the meaning of the text that it is processing and understand its semantic relationship to the query. The embedding layer generates embeddings for your text corpus and allows the RAG model to understand the meaning of the query and to generate a relevant and informative response. This is essential for a variety of tasks, such as question answering, summarisation and machine translation.

**Search and Rank Layer**

The next layer is the search and rank or the re-rank layer. The search and re-rank layer is a crucial component that is responsible for retrieving the relevant information from an external knowledge base, ranking it based on its relevance to the input query and presenting it to the generation layer for further processing. The search and re-rank layer is an essential component of RAG, as it ensures that the retrieved text is accurate, relevant and contextually appropriate. The search and re-rank layer typically consists of two components:

* A search component that uses various techniques to retrieve relevant documents from the knowledge base
* A re-rank component that uses a variety of techniques to re-rank the retrieved documents to produce the most relevant results

The search component typically uses a technique called semantic similarity. As discussed in the previous session, semantic similarity is a measure of how similar two pieces of text are in terms of their meaning. The search component uses semantic similarity to retrieve documents from a knowledge base that are relevant to the user's query.

The re-rank component of the search typically uses a variety of techniques to re-rank the retrieved documents. These techniques can include the following:

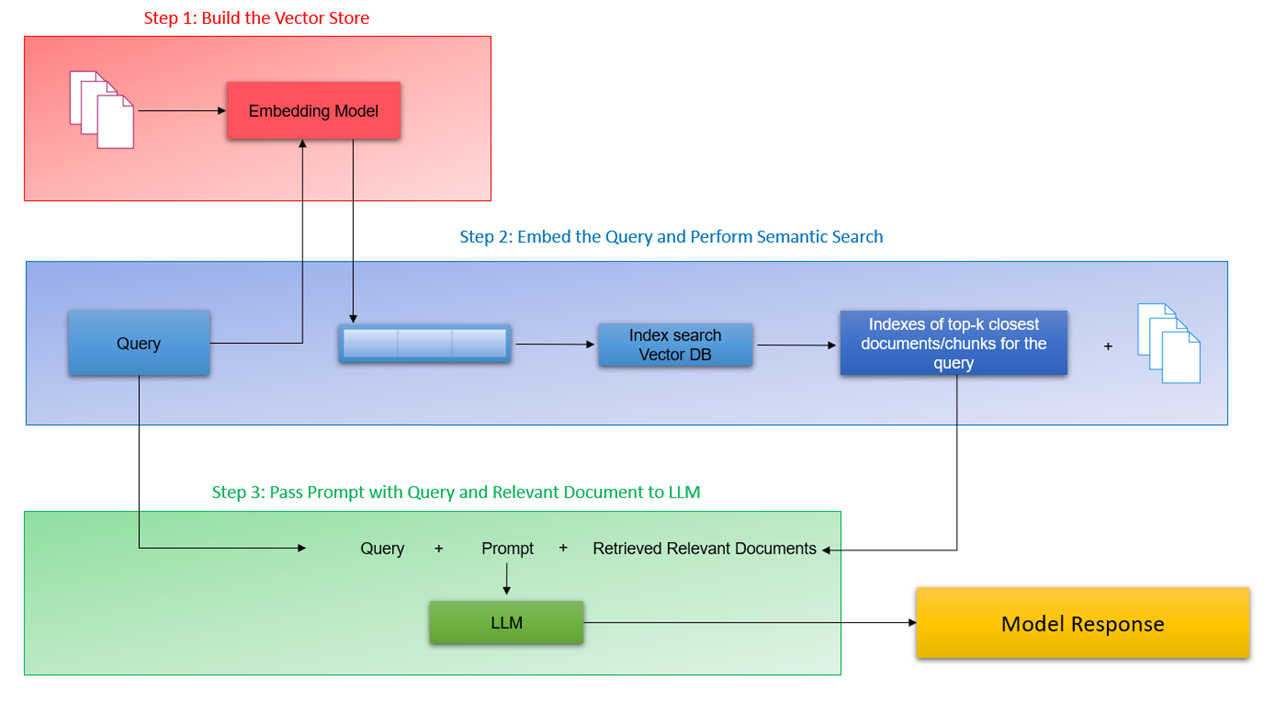
* **Ranking by relevance**: The re-rank component can rank the retrieved documents based on how relevant they are to the user's query.
* **Ranking by popularity**: The re-rank component can rank the retrieved documents based on how popular they are, such as by measuring the number of times they have been viewed or shared.
* **Ranking by freshness**: The re-rank component can rank the retrieved documents based on how recent they are, such as by measuring the date on which they were published.

The search and re-rank layer is an important part of RAG models because it allows the model to retrieve and re-rank relevant documents from a knowledge base. This is essential for numerous tasks, such as question answering, summarisation and machine translation. The search and re-rank layer is a powerful tool that can be used to improve the performance of a variety of AI tasks. It is an essential part of RAG models, and it plays a key role in helping these models retrieve and re-rank relevant information. The retrieval-based model is used to find relevant information from existing information sources. The re-rank layer is used to rank the retrieved information based on its relevance to the input query.

**Generation Layer**

The generation layer is typically the last layer of a RAG model which consists of a foundation large language model that is trained on a massive data set of text and code. As the name suggests, the generation layer allows the model to generate new text in response to a user's query. The generative model takes the retrieved information, synthesises all the data and shapes it into a coherent and contextually appropriate response. This is essential for many tasks, such as question answering, summarisation machine translation and also generative search specifically RAG. In the context of search, this layer excels in providing context and natural language capabilities for generative search.

**System Architecture**



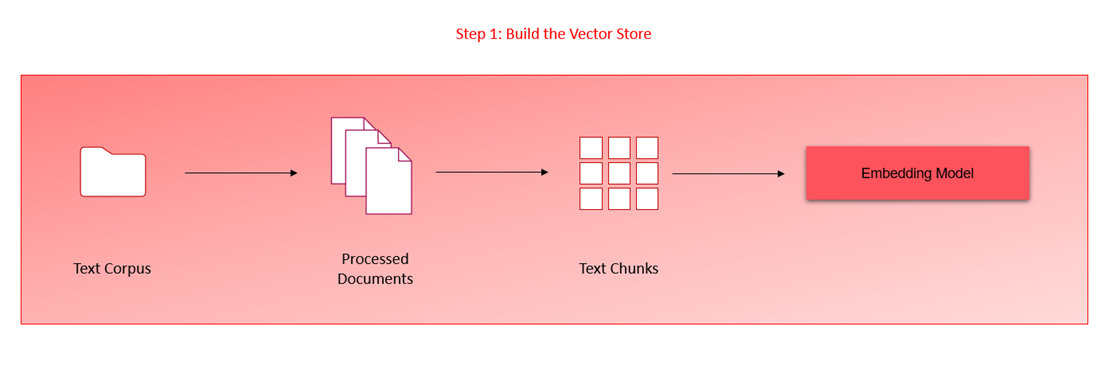
**Step 1: Build the vector store:** The first step is to build a vector store that can store documents along with metadata. A vector store is a database that stores embeddings of text data in a vector space. The documents are converted to raw text and then split into chunks. Each chunk is then represented as a vector using an embedding model. The vector store is then populated with these vectors.

**Step 2: Embed the query and perform semantic search:**The next step is to embed the user query into the same vector space as the documents in the vector store. This is done using an embedding model. Once the query is embedded, a semantic search is performed to find the closest embedding from the vector store. The entries with the highest semantic overlap with the query are retrieved.

**Step 3: Pass the prompt with the query and the relevant documents to the LLM:**The final step is to pass the prompt, which is a concatenation of the query and the retrieved documents, to the LLM. The LLM generates a response based on the context of the query, the system prompt and the relevant documents passed from the search layer. The retrieved documents serve as the knowledge bank and provide the necessary context for the query to the LLM, which helps it generate a more accurate and relevant response.

**Implementation:**

**Part 1: Text Processing**



**PdfPlumber library**

[PDFPlumber library](https://pypi.org/project/pdfplumber/) is very efficient in extracting the text contents of multiple PDF documents. The library can also represent a table in a neat list of lists format that preserves the original hierarchical structure of the document.

**Chunking & Embeddings**

page-level chunking strategy to chunk the documents at a page level along with the metadata information. reasons behind implementing page-level chunking strategy for the insurance documents. The reason behind this choice is primarily due to the structure of information in the insurance documents and the context window limit of the LLM model (GPT-3.5). Finally, you also append the relevant metadata information such as document name and page number for later retrieval.

once the text in the documents has been pre-processed and chunked, the next step is to generate vector representations using a suitable text embedding model.

once the text in the documents has been pre-processed and chunked, the next step is to generate vector representations using a suitable text embedding model.

Once the embeddings have been generated, the next step is to store them in the vector database, which is ChromaDB. As covered in the previous sessions on ChromaDB, you need to first create the Chroma collections before you can start adding documents.

get\_or\_create\_collections method, which will create a collection if not already present, and fetch it from your system if it has been created and stored previously. Next, since we are using OpenAI embeddings and not Chroma's default embedding, you need to also pass your embedding function as an argument while creating the collection.

Finally, the information that includes the document list, text and metadata information is passed to the chroma collection.

**Semantic Search with Cache**

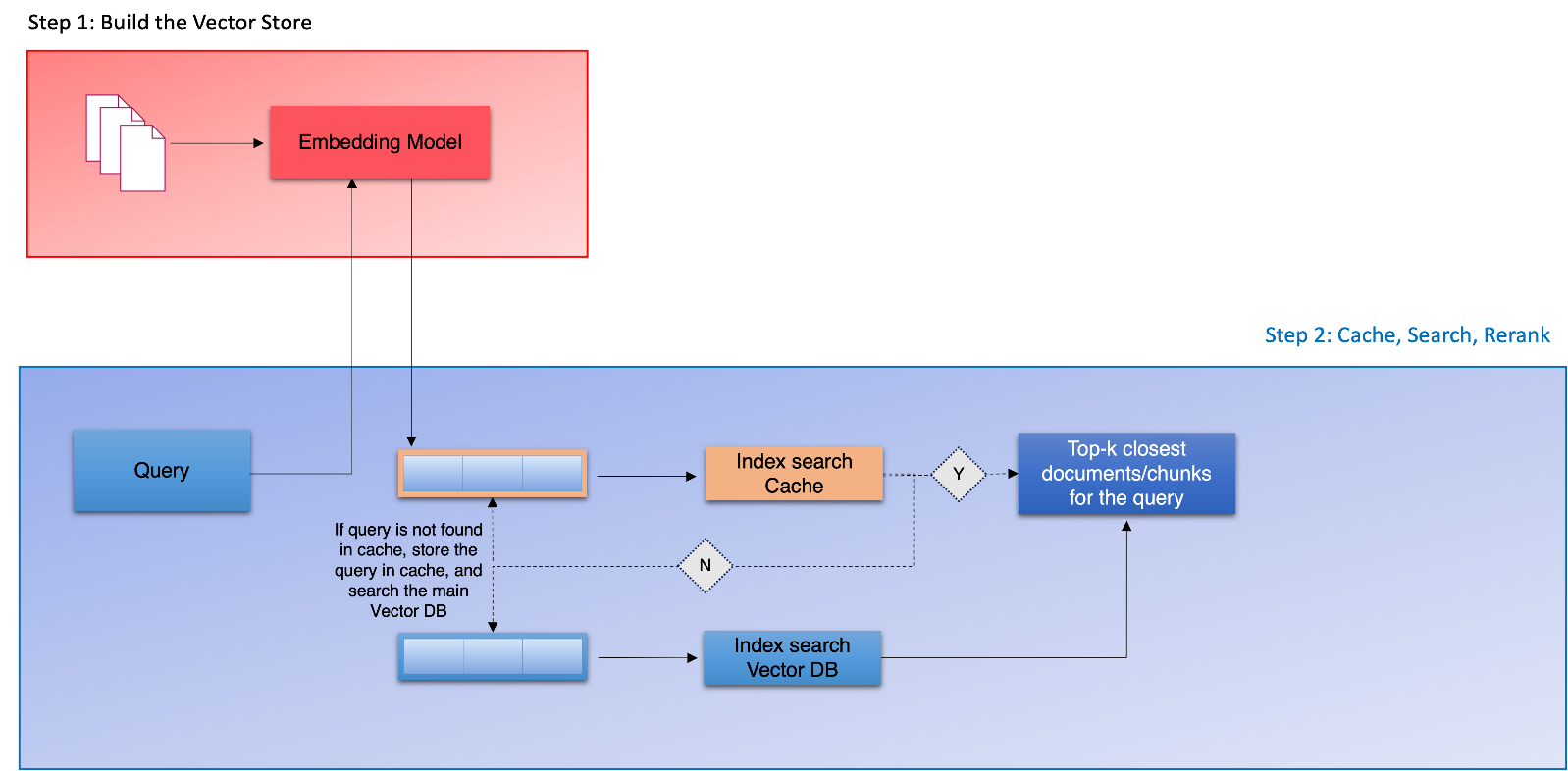
1. In the previous segment, we looked at step 1 of the RAG pipeline. We ingested the documents, processed the text and tables and generated the embeddings using a text embedding model. Once the embeddings have been generated, we then store the embeddings in a vector database such as ChromaDB.

As with any good system design, we need to consider a scenario when the application is scaled - suppose the number of documents increases or multiple users are using the application. Such a scenario opens up multiple concerns about the system’s performance

* How will the system handle multiple queries simultaneously?
* Is there scope to improve the system’s overall performance in search and retrieval?

The first concern can be solved by using vector databases and scaling up the compute units (clusters/server) for the application. For the second concern, an improvement to the overall system design is required which can be achieved by implementing a cache collection in the vector database that stores previous queries and their results in the vector database.

2. we create a cache collection within the vector database that will try to cache the queries coming in and the corresponding responses. Creating a cache is important to preserve the scalability of the RAG system, particularly when documents span in the range of 1,000 or more and multiple users are using this application concurrently. With this additional layer, when a query is first input, the system first searches within the cache collection instead of the bigger collection. Cache implementation results in an improved response time from the system since a semantic similarity search need not be performed for a query that the system has already seen. The image below shows the system design with a cache layer.



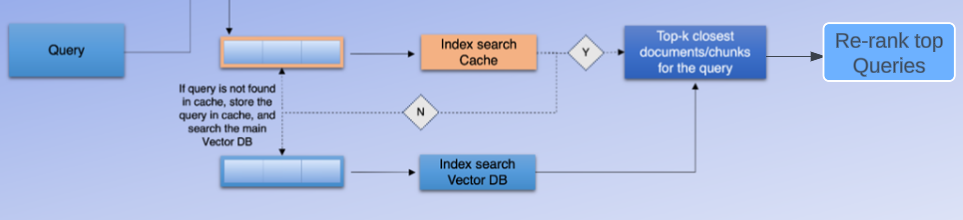
3. A semantic cache stores the meaning of a query or request instead of only the raw data along with the responses. This can reduce the number of queries the database needs to process by recalling previous queries and their results. The cache system can now circumvent the semantic search layer, which has been the bottleneck of the system, and directly provide responses for the queries that have already been generated before and stored in the cache collection. Now, when the query is passed to the application, its vector representation is generated and then searched in the cache collection first. If the query is not found in the cache collection, the system queries the main collection and finds the top k closest documents or chunks for the query. The results are then returned to the user and, simultaneously, are stored in the cache alongwith the query. Customising and monitoring the cache's performance can also make it more efficient. Since the cache stores previous queries and results, it can quickly provide the results of a query without processing it. As a result, response times can be faster, and users can experience better application performance.

**Re-Ranking**

he re-ranking stage is the next step in building the semantic search pipeline. So far, in our semantic search application, the system returns the top K documents that contain information relevant to the user’s query. The quality and accuracy of the information contained in these chunks or documents may vary - the system might retrieve documents that are not quite relevant to the search query. The purpose of the re-ranking layer is to sift through these top K results, verify the accuracy of the results in terms of the query and rank them or assign an importance score to these results for the query. Here are some of the benefits of using re-ranking in generative search:

* Improved accuracy and relevance of the generated results
* Reduced amount of irrelevant or inaccurate information presented to the user
* More personalised and informative search results
* Ability to tailor the search results to specific tasks or domains

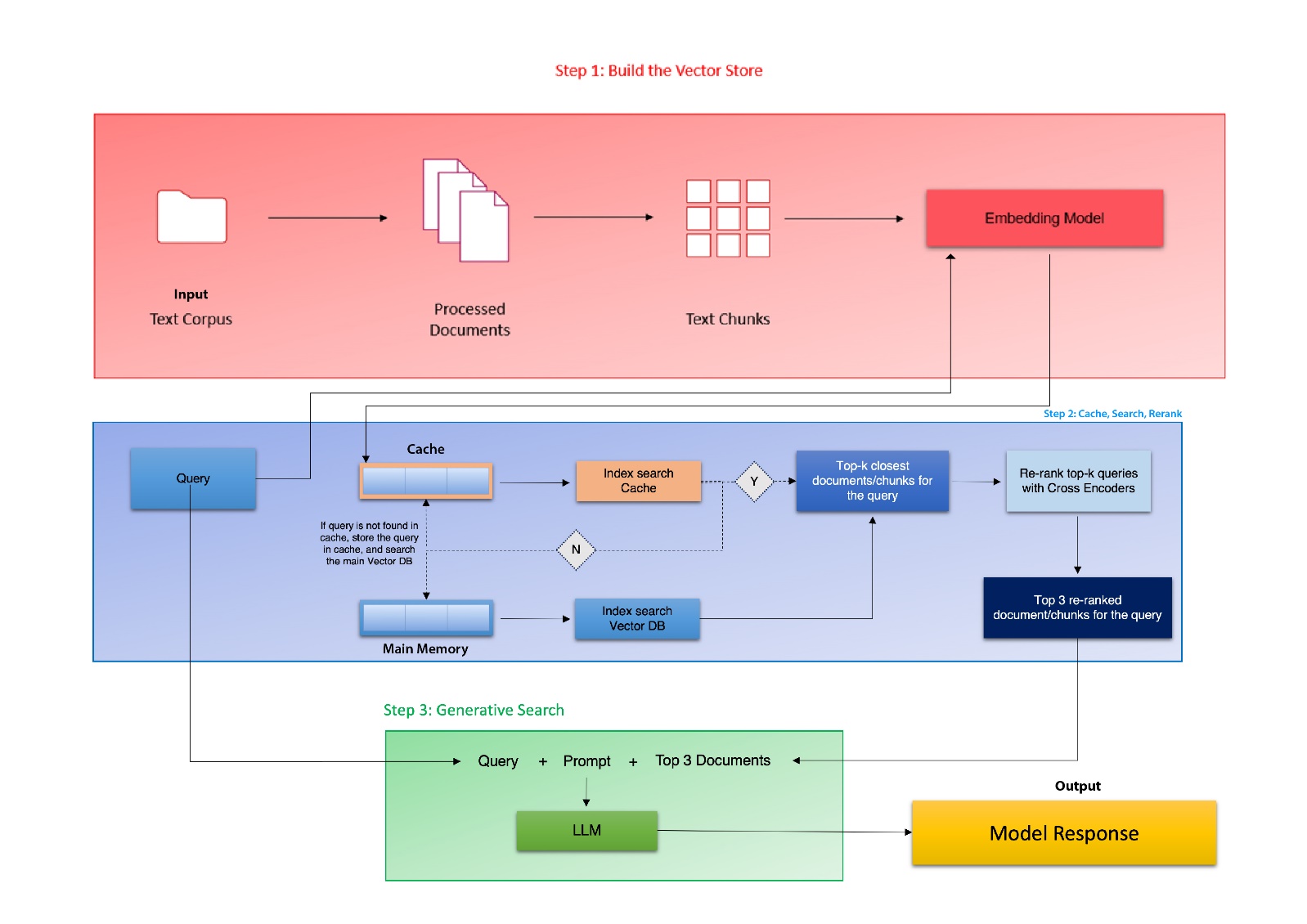
Traditionally, many methods of re-rank methods have been used in search such as Reciprocal Rank Fusion (RRF), hybrid search methods and cross-encoder models. For this project, we will focus on the popular method of using cross-encoders for our re-ranking task. The image below illustrates the re-ranking component once the search results have been collected by the semantic search layer.

****

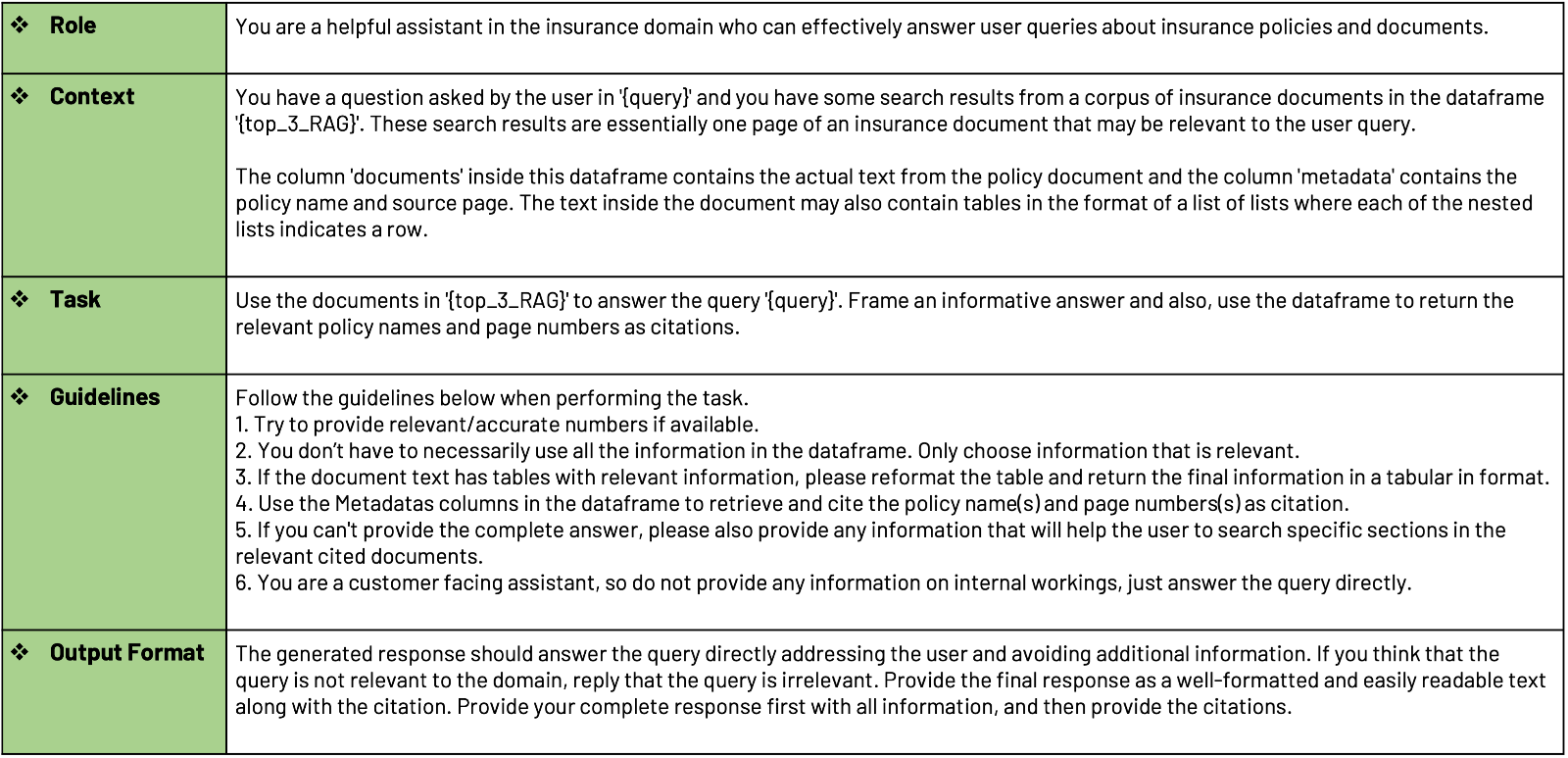
cross-encoder models are transformer models that can be used to learn the semantic similarity between two text sequences. They are trained on a large data set of text pairs, where each pair is labelled with a score indicating how similar the two sequences are. Once trained, cross-encoder models can be used to compute the similarity between any two text sequences, even if they have never been seen before. The image below illustrates the function of a cross-encoder model.

**GENRATION LAYER**

generative search application is the generation layer. This layer uses the generation capabilities of a large language model (LLM) to augment the system’s output.

 the semantic search layer of our application performs the retrieval process or the information retrieval task and returns the top k documents for the user’s query. These top k results, along with the user’s query and the system prompt, are then passed to the LLM model to generate a model response that best answers the user’s query. The LLM is provided with three inputs - query, prompt and top retrieved documents - as shown in the image below.

The retrieval system retrieves the top K results from the knowledge base for the user’s query, which are then passed to the LLM for generating the response. The LLM is provided with the system prompt that condenses the information from the top K results and generates a unique response to the query. As discussed in the video, the system prompt is shown below.



The system prompt can vary depending on the nature of the application. For generating responses for the documents pertaining to the insurance domain, the prompt above has generated results with reasonable performance. Depending on the domain and documents, the system prompt can be modified to generate responses accordingly.