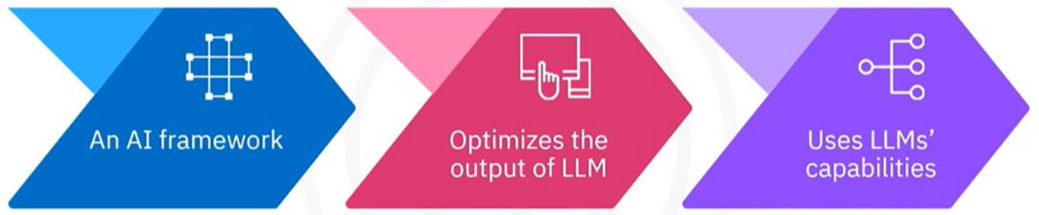
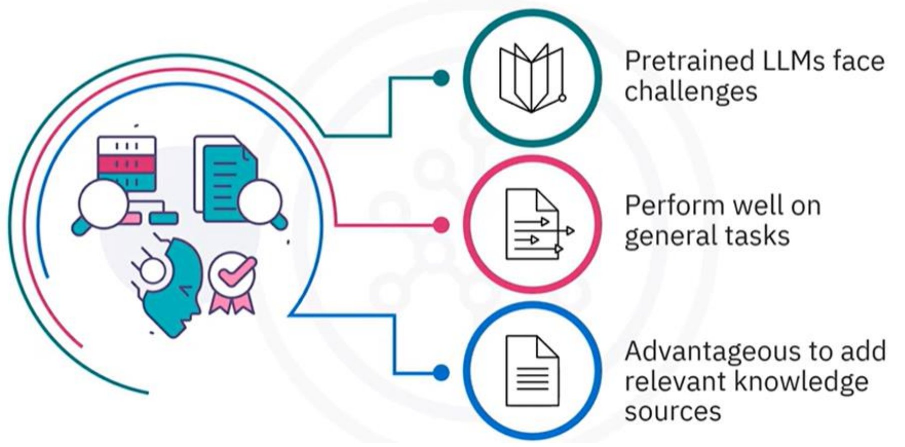
Build RAG Applications

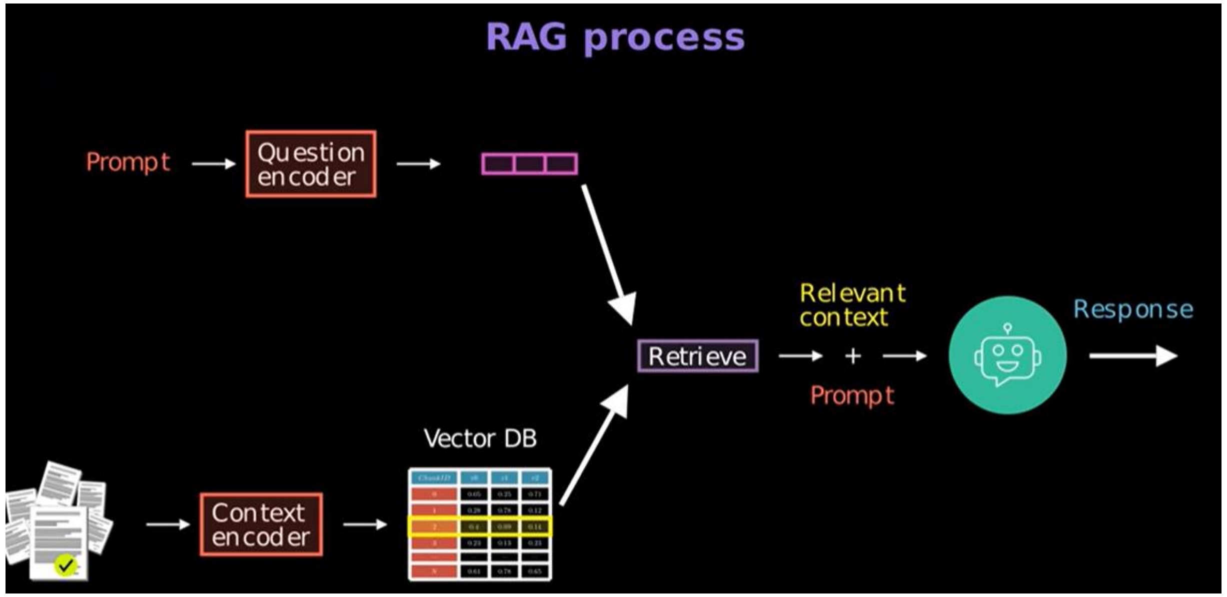
What is RAG?



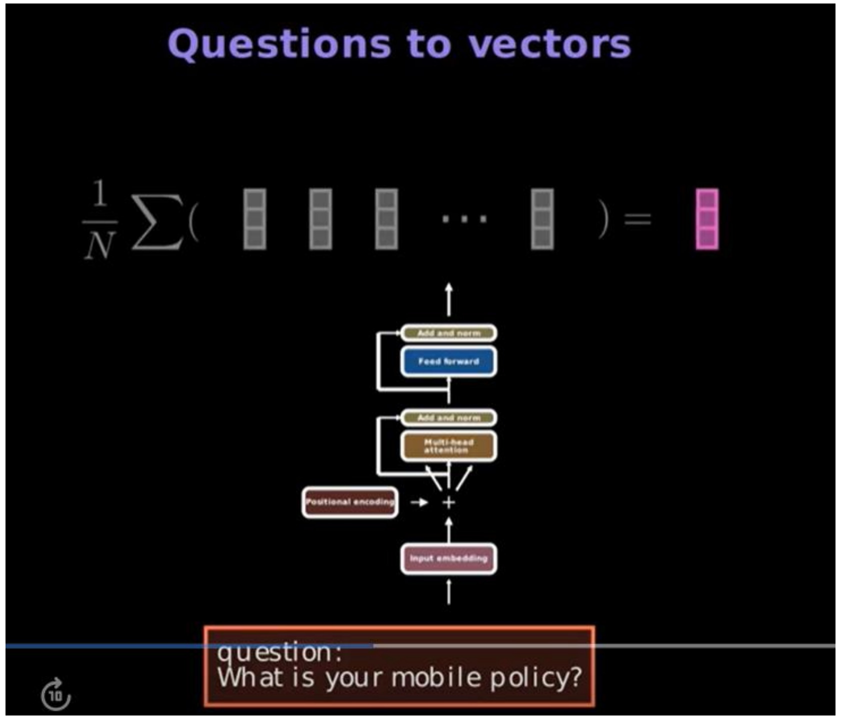
Importance of RAG in training LLMs

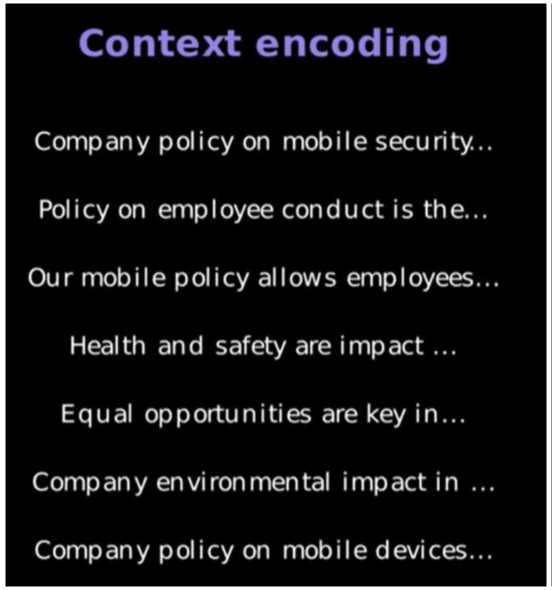

Pretrained LLMs face challenges
Perform well on general tasks
Advantageous to add relevant knowledge sources

RAG process



Questions to vectors





Context encoding

Company policy on mobile security...

Policy on employee conduct is the...

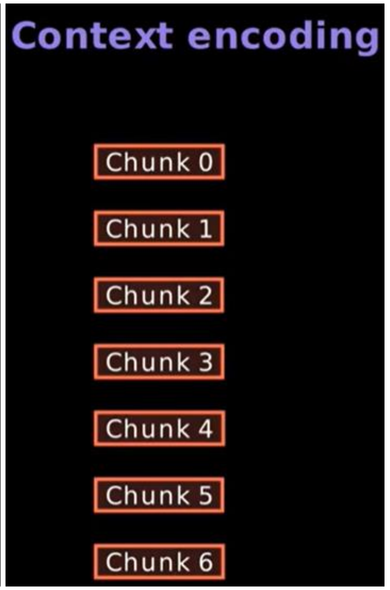
Our mobile policy allows employees...

Health and safety are impact ...

Equal opportunities are key in...

Company environmental impact in ...

Company policy on mobile devices...



Context encoding

Chunk 0

Chunk 1

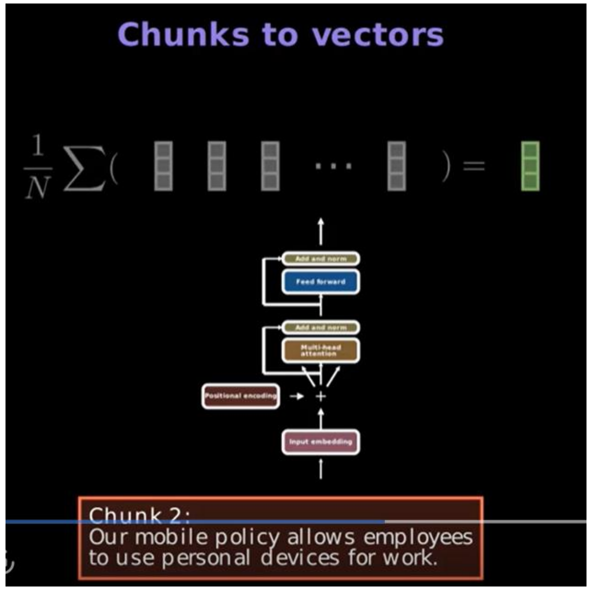
Chunk 2

Chunk 3

Chunk 4

Chunk 5

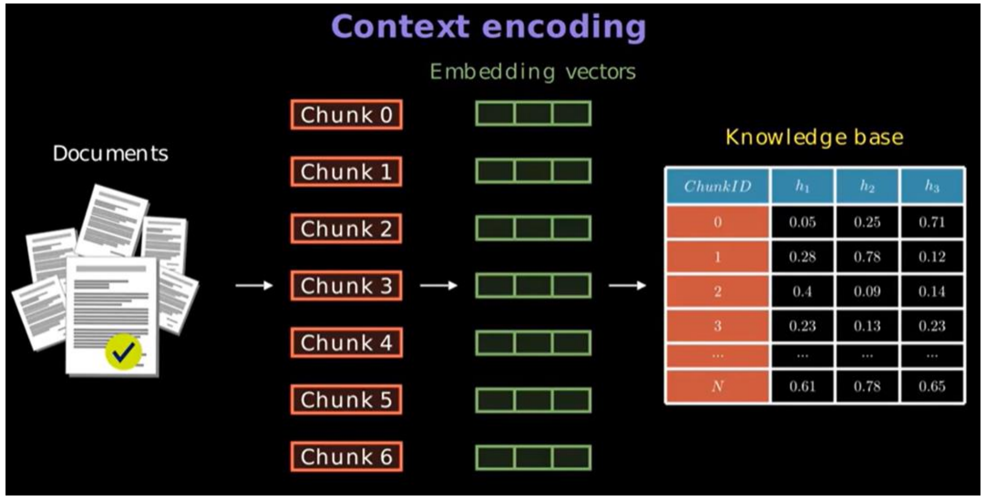
Chunk 6



Chunks to vectors

$$\frac{1}{N}\sum(\ \cdots\ ) =$$

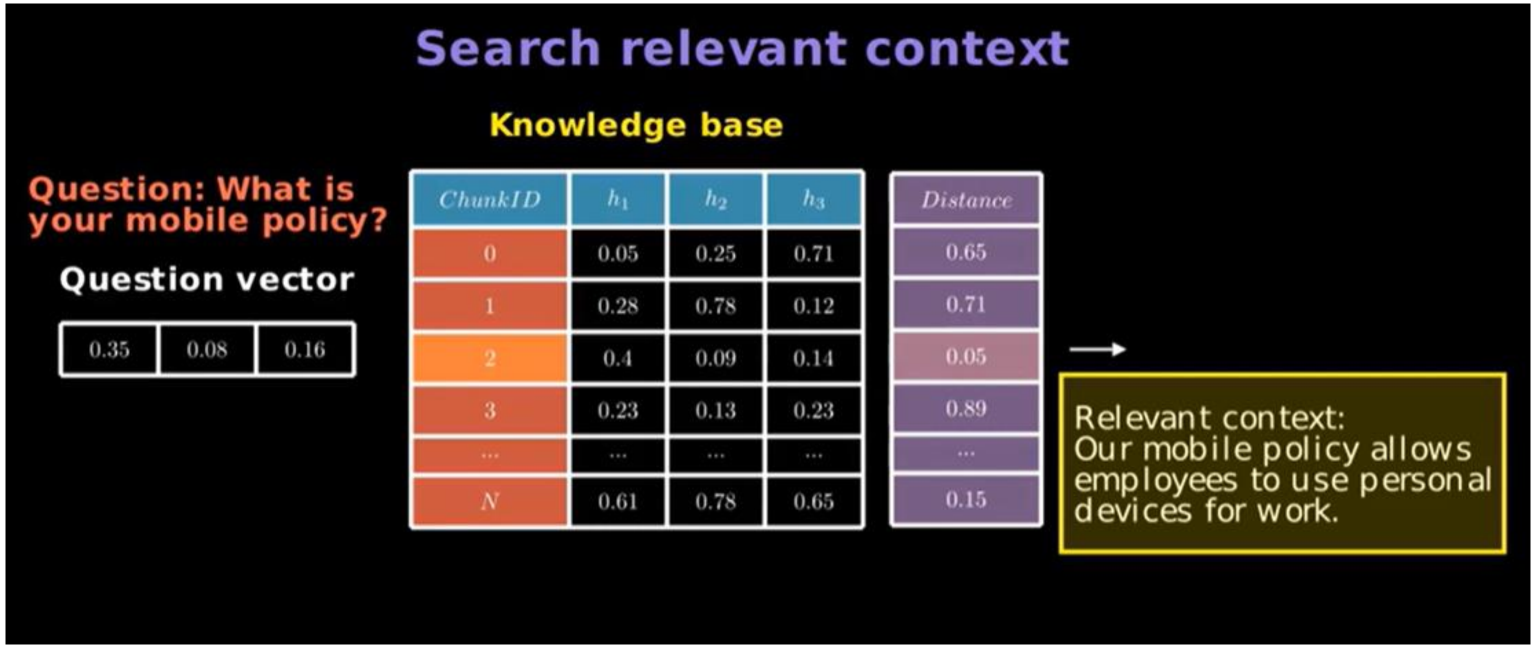
Chunk 2:
Our mobile policy allows employees to use personal devices for work.



Context encoding

Knowledge base

$ChunkID$	h_1	h_2	h_3
0	0.05	0.25	0.71
1	0.28	0.78	0.12
2	0.4	0.09	0.14
3	0.23	0.13	0.23
...	...	...	...
N	0.61	0.78	0.65



Search relevant context

Question: What is your mobile policy?

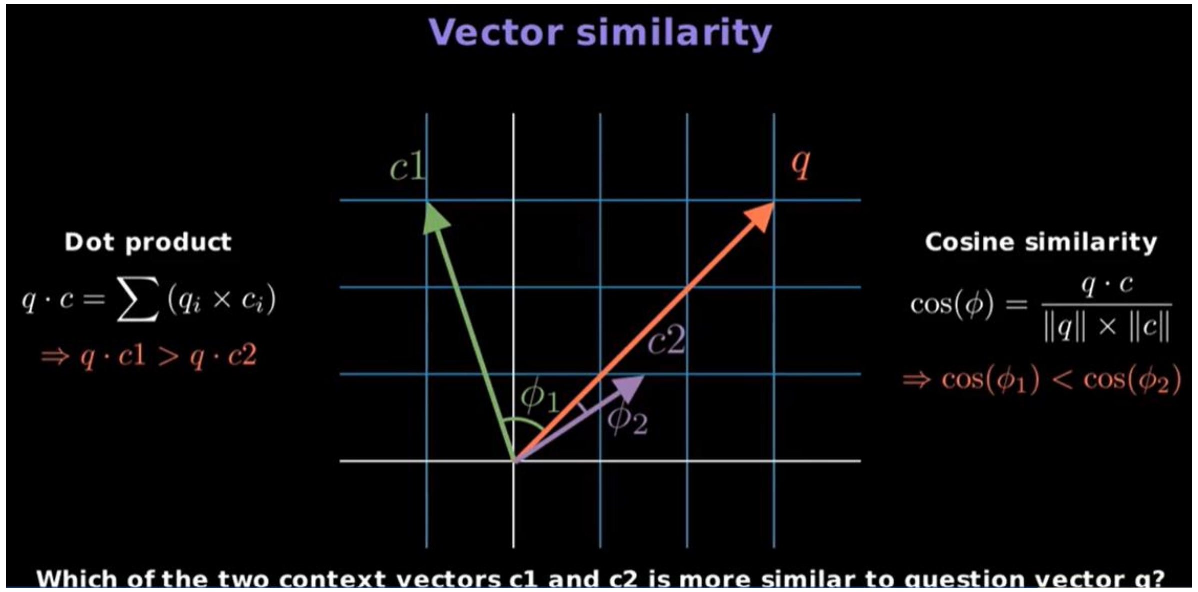
Question vector

0.35	0.08	0.16

Knowledge base

$ChunkID$	h_1	h_2	h_3	$Distance$
0	0.05	0.25	0.71	0.65
1	0.28	0.78	0.12	0.71
2	0.4	0.09	0.14	0.05
3	0.23	0.13	0.23	0.89
...	...	...	...	...
N	0.61	0.78	0.65	0.15

Relevant context:
Our mobile policy allows employees to use personal devices for work.



Vector similarity

Dot product

$$q \cdot c = \sum (q_i \times c_i)$$

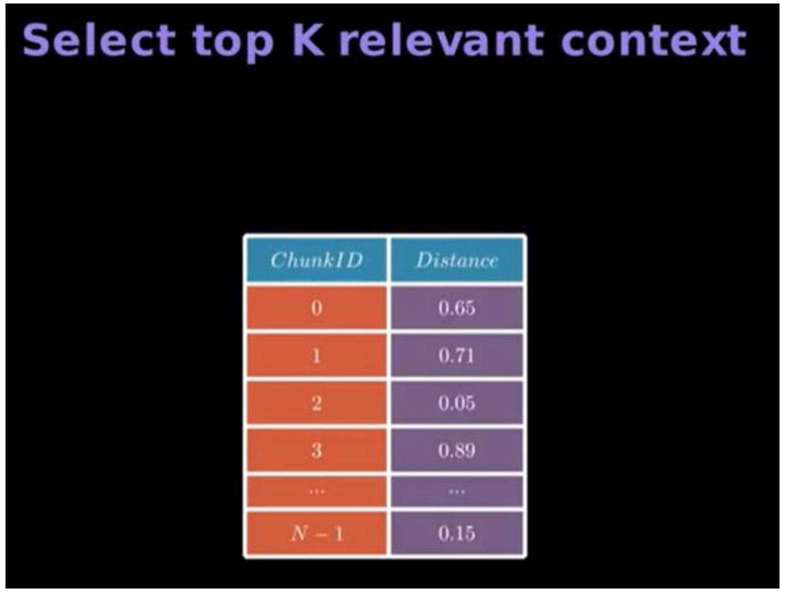
$$\Rightarrow q \cdot c1 > q \cdot c2$$

Cosine similarity

$$\cos(\phi) = \frac{q \cdot c}{\|q\| \times \|c\|}$$

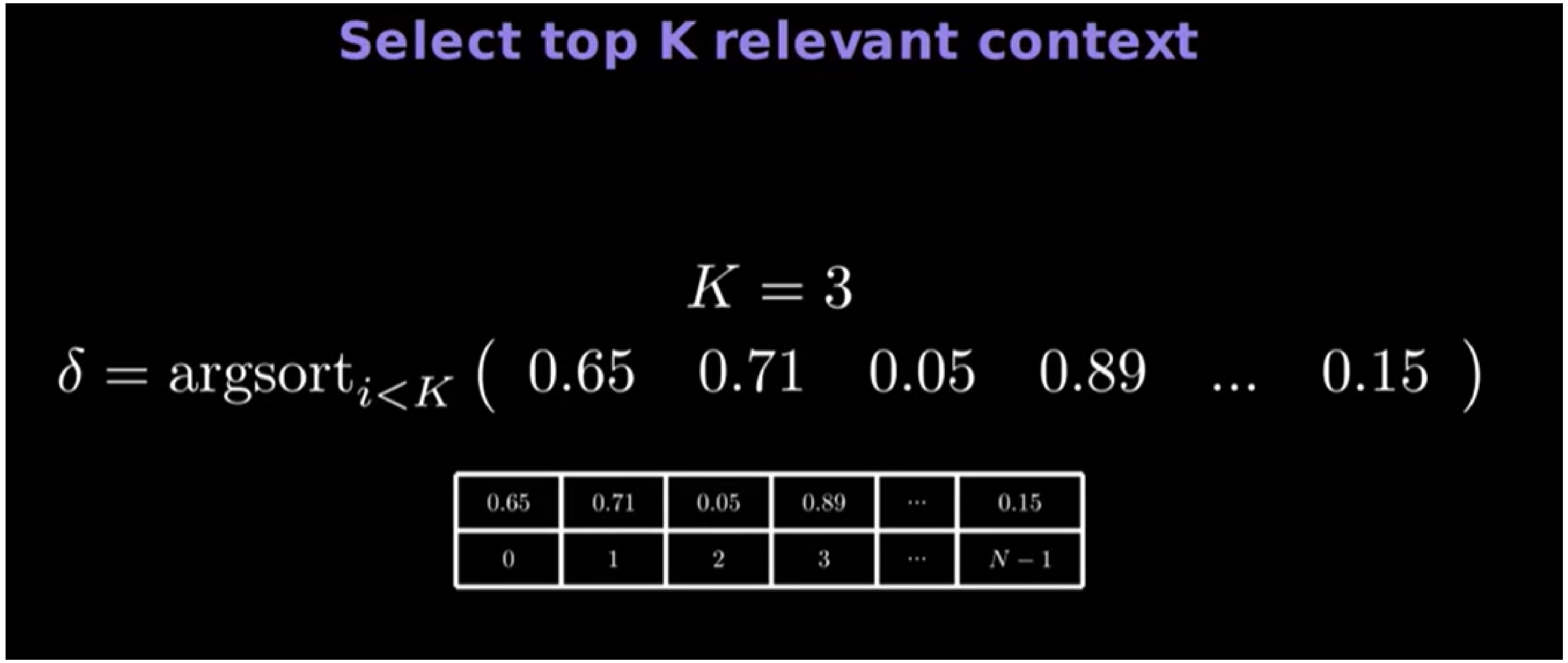
$$\Rightarrow \cos(\phi_1) < \cos(\phi_2)$$

Which of the two context vectors c1 and c2 is more similar to question vector q?



Select top K relevant context

ChunkID	*Distance*
0	0.65
1	0.71
2	0.05
3	0.89
...	...
$N-1$	0.15

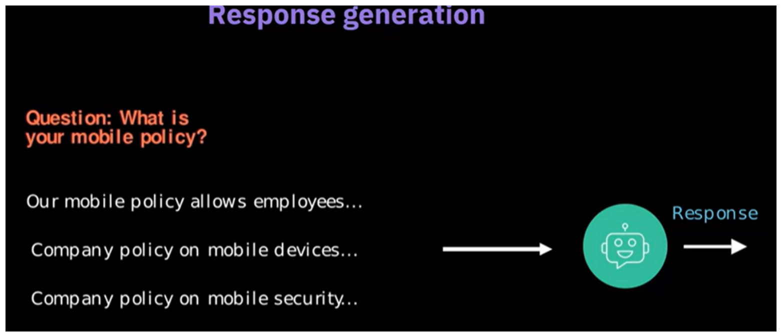


Select top K relevant context

$$K = 3$$

$$\delta = \text{argsort}_{i<K} \left(\, 0.65 \quad 0.71 \quad 0.05 \quad 0.89 \quad \ldots \quad 0.15 \, \right)$$

0.65	0.71	0.05	0.89	...	0.15
0	1	2	3	...	$N-1$



Response generation

Question: What is your mobile policy?

Our mobile policy allows employees...

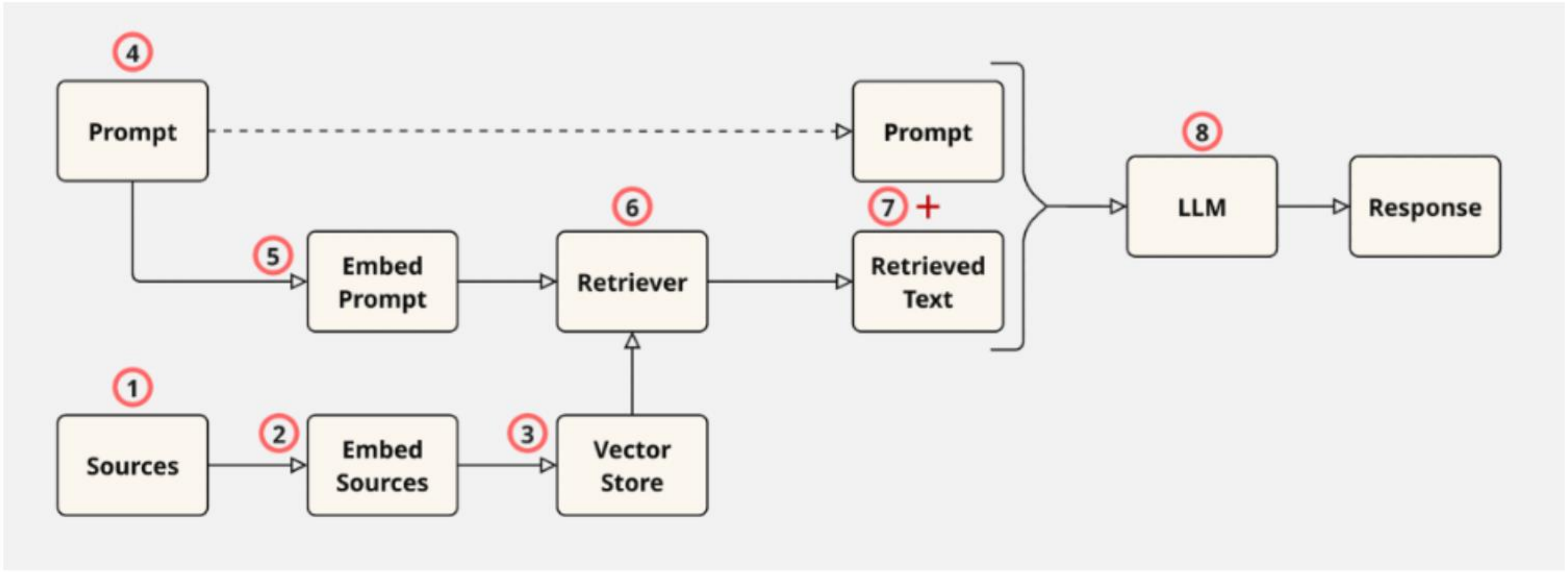
Company policy on mobile devices...

Company policy on mobile security...

Recap

- RAG helps generate responses
- Challenging for chatbot to generate responses for specific domains such as the company's mobile policy
- To generate responses, a chatbot:
  - Encodes inserted question or prompt
  - Breaks down into smaller chunks of text
  - Converts text chunks into high-dimensional vectors using distance metrics
  - Selects a vector closer to the text chunks from the knowledge base to generate a relevant response

The following diagram illustrates the RAG process for a basic RAG system. Note that this is just one possible representation, and alternative diagrams may result from various modifications or adaptations of the RAG system. However, this diagram captures the core concept, as all variations build on the common themes presented here:



What about models with long context lengths?

RAG was developed and gained popularity during a period when models with large context lengths were uncommon. Today, models capable of handling context lengths up to 128,000 tokens or more are widely available. To put this into perspective, a token is a unit of text that can represent a word, part of a word, or even punctuation marks and spaces. Since tokens are not strictly equivalent to words, the ratio of tokens to words can vary. In English, a good estimate is that 100 tokens correspond to about 75 words. Based on this, a model with a 128,000-token context length can process an English text of approximately 96,000 words. This is long enough to include numerous relevant details within a prompt, providing the model with substantial contextual information.

However, relying solely on such extended context lengths presents several limitations:

1. **Input Dependency:** Users must already possess the necessary source information to provide within the prompt. Without this, the model cannot generate insights or solutions.
2. **Limited Capacity:** Although a 128,000-token capacity is significant, it may still fall short for extremely lengthy texts. For example, English translations of War and Peace by Leo Tolstoy generally exceed 560,000 words—more than five times this limit.
3. **Redundancy Issues:** Even when all relevant details fit within the prompt, irrelevant or repeated information can dilute the LLM's focus. This creates a "needle in a haystack" challenge for the model, as it must sift through vast data to extract critical facts.
4. **Processing Time:** Longer prompts require additional processing time. Since AI models analyze input by breaking it into tokens, more tokens result in longer processing times.
5. **Cost Implications:** Using a high number of tokens in prompts also increases both computational and financial costs, making this approach potentially less practical in some applications.

RAG helps address these challenges, either partially or fully:

1. **Input Dependency:** RAG connects to an external data store that users do not need to provide or even be aware of to interact successfully with the system.
2. **Limited Capacity:** RAG retrieves only the text most relevant to the prompt, creating smaller augmented prompts that fit within LLM context limits.
3. **Redundancy Issues:** RAG ensures only the most pertinent information is passed to the LLM, making it easier for the system to identify relevant details in source texts.
4. **Processing Time:** Shorter augmented prompts in RAG reduce the response generation time compared to non-RAG systems that include all available source information within the augmented prompt.
5. **Cost Implications:** By using shorter augmented prompts, RAG reduces response generation costs, especially for systems with extensive data sources.

- Retrieval-Augmented Generation (RAG) is a machine-learning technique that integrates information retrieval with generative AI to produce accurate and context-aware responses.
- RAG enhances Large Language Models (LLMs) by integrating external or domain-specific knowledge without retraining. This helps LLMs generate more accurate and contextually relevant responses for specialized queries, such as a company's mobile policy.
- RAG consists of two main components: the Retriever, which extracts relevant data from a knowledge base, and the Generator, which uses the retrieved information to generate responses in natural language.
- The RAG process comprises four steps: Text Embedding, Retrieval, Augmented Query Creation, and Model Generation.
- Text Embedding converts user prompts and knowledge base documents into high-dimensional vectors using AI models such as BERT or GPT.
- Retrieval matches the user query with similar vectors from the knowledge base to retrieve relevant information.
- Augmented Query Creation combines retrieved content with the user prompt.
- Model Generation uses the created augmented query to generate a response using the content from the knowledge base.
- Prompt encoding converts a text-based prompt into a numerical representation that an LLM can process. It uses Token Embedding and Vector Averaging to break documents into smaller text chunks, convert them into vectors, and index them in a vector database.

- Distance Metrics measure similarity between user queries and document vectors using methods such as dot product (magnitude-based) or cosine similarity (direction-based).
- RAG is an efficient response generation technique. It retrieves the most relevant text chunks from the knowledge base to augment the model's knowledge and produce an informed response. This ensures responses are accurate, domain-specific, and up-to-date.
- RAG allows chatbots to provide specialized answers by integrating relevant external knowledge, making them more reliable for industry-specific or confidential topics.

Gradio

How does Gradio work?

Write Python code to define functions for application

Use Gradio to create an interface
- Use Gradio's interface class to specify inputs and outputs
- Configure interface to define user interaction

Launch Gradio server using the launch method
- Start a local server on machine
- Create a web interface

Access web interface through local or public URL
- Users interact with interface in real time

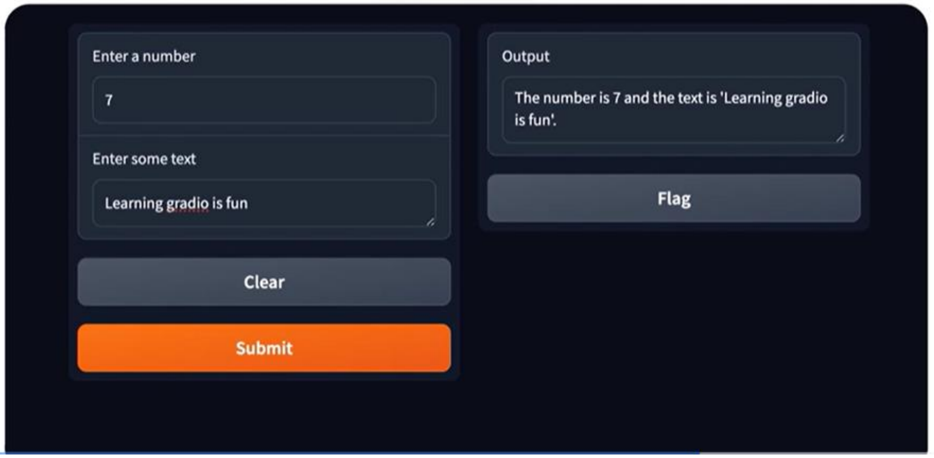
Simple text input and output with Gradio

```
def process_text(text):
      return f"You entered: '{text}'"

demo = gr.Interface(
      fn = process_text,
      inputs = gr.Textbox(label = "Enter some text"), # Textbox input
      outputs = gr.Textbox(label = "Output") # Textbox output
)

demo.launch()
```

Multiple inputs with Gradio



Enter a number

7

Enter some text

Learning gradio is fun

Output

The number is 7 and the text is 'Learning gradio is fun'.

Upload/Drop files with Gradio

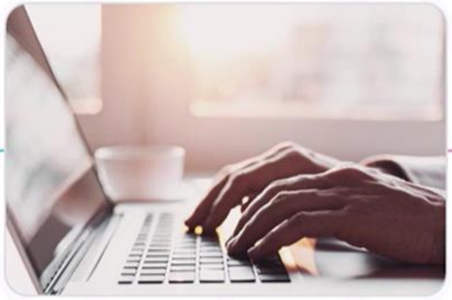
```
def count_files(files):
      return f"Number of files uploaded: {len(files)}"

demo = gr.Interface(
      fn = count_files,
      inputs = gr.File(file_count = "multiple",
                       type = "filepath",
                       label = "Upload or Drag Files Here"),
      outputs = gr .Textbox(label = "Number of Files Uploaded"),
)
```

High-level overview of LlamaIndex

LlamaIndex

- Is a framework for building LLM-powered context augmentation



Context augmentation

- Is the process of making data available to the LLM
- Allows the LLM to perform a specific task while grounding the LLM's response in the provided context

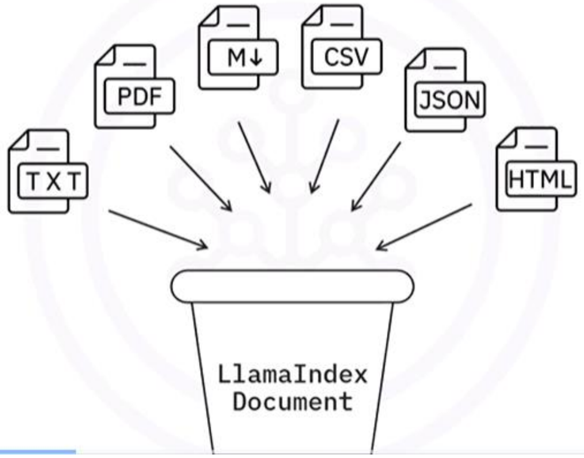
Typical use cases

Question-Answering with Retrieval-Augment Generation (RAG)

Chatbots extend the basic RAG pipeline

Document understanding and data extraction

Loading source documents



The LlamaIndex Document class

Creates an object of the Document class

Calls the Document object's dict method

The Document object has several key components:

- An ID that uniquely identifies the document
- A placeholder for an embedding
- A metadata dict for metadata storage

```python
from llama_index.core import Document
mydocument = Document(text="Hello LlamaIndex")
mydocument.dict()
```

```
{'id_': '8b144f47-8267-4245-995e-65876c3f04a0',
 'embedding': None,
 'metadata': {},
 'excluded_embed_metadata_keys': [],
 'excluded_llm_metadata_keys': [],
 'relationships': {},
 'metadata_template': '{key}: {value}',
```

The LlamaIndex Document class

The Document object has several key components:

- A relationships dict for linking to other documents
- The text present in the document

```
from llama_index.core import
Document
mydocument = Document(text="Hello LlamaIndex")
mydocument.dict()
```

```
 'excluded_embed_metadata_keys': [],
 'excluded_llm_metadata_keys': [],
 'relationships': {},
 'metadata_template': '{key}: {value}',
 'metadata_separator': '\n',
 'text_resource': {'embeddings': None,
  'text': 'Hello LlamaIndex',
```

Using SimpleDirectoryReader

- Import `SimpleDirectoryReader`

```
from llama_index.core import SimpleDirectoryReader
```

- Load all the files in `my_folder`

```
documents = SimpleDirectoryReader("my_folder").load_data()
```

- Load all the files in `my_folder` and all its subdirectories

```
documents = SimpleDirectoryReader("my_folder", recursive=True).load_data()
```

Using SimpleDirectoryReader

- Load specific files using the `input_files` parameter

```
documents = SimpleDirectoryReader(
    input_files=["my_folder/some_text.txt"]
).load_data()
```

- Load specific file types using the `required_ext` parameter

```
documents = SimpleDirectoryReader(
    input_dir="my_folder",
    required_exts=[".txt", ".csv"]
).load_data()
```

Using LlamaIndex nodes

- LlamaIndex nodes are simply text chunks
- LlamaIndex provides an easy-to-use text chunker called `SentenceSplitter`.

```
from llama_index.core.node_parser import SentenceSplitter
```

- `SentenceSplitter` splits long texts recursively based on specific characters, such as newline characters and periods.

Using LlamaIndex nodes

To use SentenceSplitter:

- Import sentence `SentenceSplitter` from `llama_index.core.node`:

```
from llama_index.core.node_parser import SentenceSplitter
```

- Define the node parser as an instance of `SentenceSplitter`
  - Pass the `chunk_size` parameter to control the max length of chunks
  - Pass in the chunk_overlap parameter to control the max overlap between chunks

```
node_parser = SentenceSplitter(chunk_size=1024, chunk_overlap=20)
```

Using LlamaIndex nodes

To use SentenceSplitter:

- Use the `get_nodes_from_documents` method to split documents into individual nodes

```
nodes = node_parser.get_nodes_from_documents(
    documents,
    show_progress=False
)
```

- The node parser returns a list of LlamaIndex TextNode instances, which are similar in structure to Document instances

Chunk Documents using other splitters

A semantic splitter
A wrapper around any LangChain splitter

Recap

- LlamaIndex is a framework for building LLM-powered context augmentation.
- Use cases for LlamaIndex include question-answering using RAG for chatbots, document understanding, and data extraction.
- LlamaIndex loads source documents by converting text, PDF, Markdown, CSV, JSON, and HTML files into LlamaIndex `Document` objects.
- The LlamaIndex `Document` class creates a document object with several key components, including a unique ID, an embedding placeholder, a metadata dictionary, a relationships dictionary for linking to other documents, and the text in the source document.

- You can use the `SimpleDirectoryReader` to import and load files from a folder, its subdirectories, specific files, or specific file types.
- You can use LlamaIndex's `SentenceSplitter` to recursively split documents into chunks based on predefined separators.
- You can also chunk documents using other splitters, such as a semantic splitter, or by using a wrapper around any LangChain splitter.

Generate and store embeddings in-memory

LlamaIndex uses the **`VectorStoreIndex`** class to generate and store embeddings

For simple use cases where vectors are generated using a default model and stored in-memory:

```
from llama_index.core import VectorStoreIndex
index = VectorStoreIndex(nodes)
```

Generate and store embeddings persistently

For complex cases with custom models and persistent storage:

1. Import libraries:

```
import chromadb
from llama_index.core import VectorStoreIndex, StorageContext
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
```

Generate and store embeddings persistently

For complex cases with custom models and persistent storage:

2. Define the embedding model:

```
embed_model = HuggingFaceEmbedding(
    model_name="sentence-transformers/all-MiniLM-L6-v2"
)
```

Generate and store embeddings persistently

For complex cases with custom models and persistent storage:

3. Set up the vector store and storage context:

```
db = chromadb.PersistentClient(path="chroma_db_test")
chroma_collection = db.get_or_create_collection("mycollection")
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(
    vector_store=vector_store
)
```

Generate and store embeddings persistently

For complex cases with custom models and persistent storage:

4. Pass the nodes, the model, and storage context to **`VectorStoreIndex`**:

```
index = VectorStoreIndex(
    nodes=nodes,
    embed_model=embed_model,
    storage_context=storage_context
)
```

Prompt embedding and retrieval steps

1. Create the retriever by calling the **`as_retriever`** method on the **`VectorStoreIndex`** object.
2. Pass the user's prompt to the retriever object's **`retrieve`** method.

```
retriever = index.as_retriever()
retrieved_nodes = retriever.retrieve(
    "User's prompt"
)
```

Alternatively, to specify the maximum number of items retrieved:

1. Pass the maximum number of items to retrieve to the **`similarity_top_k`** parameter when creating the retriever.
2. The Retriever retrieves the *k* nodes most similar to the user's query, in this case, 5 nodes.

```
retriever =
index.as_retriever(similarity_top_k=5)
retrieved_nodes = retriever.retrieve(
    "User's prompt"
)
```

LlamaIndex's response synthesizer

- LlamaIndex uses a **response synthesizer** to combine prompt augmentation, LLM querying, and response regeneration.
- Given a user's prompt and retrieved nodes, the response synthesizer's **`synthesize`** method generates a response from the LLM.

```
from llama_index.core import get_response_synthesizer
response_synthesizer = get_response_synthesizer()
response = response_synthesizer.synthesize(
    query="User's prompt", nodes=retrieved_nodes
)
```

LlamaIndex's query engine

- LlamaIndex's **query engine** combines the prompt embedding, retrieval, prompt augmentation, LLM querying, and response generation steps.
- Given a user's prompt, the query engine's query method generates a response from the LLM.

```
query_engine = index.as_query_engine()
response = query_engine.query("User's prompt")
```

LlamaIndex's query engines

Tip! Modify a query engine by:

- Changing the default LLM
- Defining a custom prompt template
- Specifying a custom retriever

Recap

- LlamaIndex uses the `VectorStoreIndex` class for embedding generation and storage.
- You create the retriever by calling the `as_retriever` method on the `` `VectorStoreIndex` `` object. Then, you pass the user's prompt to the retriever object's `` `retrieve` `` method to obtain relevant results.
- The response synthesizer in LlamaIndex combines prompt augmentation, LLM querying, and response generation in one step.
- The response synthesizer uses the original prompt and retrieved nodes to generate a response, refining it if needed with leftover nodes.
- You can customize response synthesizer behavior using different LLMs or prompt templates.

Recap

- A query engine can compress the process further by combining prompt embedding, retrieval, prompt augmentation, LLM querying, and response generation RAG steps into a few simple commands.
- A query engine can be modified by changing the default LLM, customizing the prompt template, or by modifying the retriever.

Document chunking

LangChain and LlamaIndex offer a variety of chunking strategies:

- LangChain
  - For typical use cases, LangChain offers length-based text splitters. These come in two types:
    - A character-based text splitter, called `CharacterTextSplitter`, divides text using a specific character sequence, such as \n\n for line breaks. Additionally, the chunk sizes are limited by a settable maximum character length.
    - Token-based text splitters limit the maximum chunk length by a settable number of tokens. One example is `TokenTextSplitter`, which encodes text into tokens, splits the tokens into chunks based on length, and then decodes the chunks back into text. Additionally, `CharacterTextSplitter` can also be used for token-based text splitting. In this context, `CharacterTextSplitter` splits text based on a specific character sequence but limits the chunk size by the maximum number of tokens instead of character length.
  - An extension of some of the above concepts is provided by the `RecursiveCharacterTextSplitter`. `RecursiveCharacterTextSplitter` splits on characters from a specific list. For instance, if the list is `[“\n\n”, “.”]`, the text is first split on two newline characters. Then, if any of the resulting chunks are too long, those chunks are further split on the next character in the list, in this case, a period. Hence, the recursive nature of the text splitting.
  - Document-structured text splitters split documents based on some of the inherent elements in the files. For instance, the `MarkdownHeaderTextSplitter` splits on markdown headings, such as `#`, `##` etc. LangChain provides document splitters for markdown files, code, HTML, and a recursive splitter for JSON.
  - There is also a semantic meaning-based splitter called `SemanticChunker` that splits text if the similarity between two sentences is below a certain threshold. This can be useful for finding natural breaks between concepts in text.

- LlamaIndex
  - First off, it's worth noting that LlamaIndex refers to a text chunk as a `node`.
  - For basic use cases, LlamaIndex provides a `SentenceSplitter`, which works similarly to LangChain's `RecursiveCharacterTextSplitter` except that it is token-based in the sense that the chunk size is defined by the number of tokens.
  - LlamaIndex provides several file-based "node parsers" that are similar to LangChain's document-structured text splitters. There are splitters for HTML, JSON, and markdown files. There is also a code splitter, though in contrast to LangChain, the code splitter is text-based, not file-based.
  - `SemanticSplitterNodeParser` provides similar functionality to LangChain's `SemanticChunker` and splits text if the similarity between two sentences falls below a threshold.
  - Finally, LlamaIndex provides a `LangChainNodeParser` which wraps around any LangChain text splitter. This allows you to use any splitter from LangChain in LlamaIndex!

Once again, for basic usage LlamaIndex typically provides a slightly better experience with its `SentenceSplitter` being set up by default with a more comprehensive splitting list than LangChain's `RecursiveCharacterTextSplitter`. However, both frameworks provide comprehensive coverage of splitting strategies, with LlamaIndex even providing a wrapper for LangChain's splitters if you prefer those while using the other aspects of LlamaIndex.

3. Store Vectors in a Vector Store

This section explores the differences between LangChain and LlamaIndex with respect to storing the vector embeddings.

- LangChain
  - LangChain does not have a single, all-encompassing vector store class, and typically relies on integrations with external libraries to integrate a vector database. The only commonly used vector store defined within the LangChain core library is the `InMemoryVectorStore` which stores vectors in memory. Integrations for external libraries and full-fledged vector databases include:
    - `Chroma` for integration with Chroma DB
    - `FAISS` for integration with the Facebook AI Similarity Search (FAISS) library and vector database
    - `Milvus` for integration with the Milvus vector database
    - `PGVector` for integration with `pgvector`-extended PostgreSQL
- LlamaIndex
  - LlamaIndex provides a powerful `VectorStoreIndex` class that stores vectors in memory by default but can be integrated with full-fledged vector databases such as Chroma DB or Faiss. Thus, LlamaIndex's `VectorStoreIndex` class ensures ease of use because the vector store backend can be swapped out without changing any of the downstream code.
  - Moreover, as opposed to the typical workflow in LangChain, in LlamaIndex the vectors are embedded and stored in the vector store with just one command: `index = VectorStoreIndex(nodes)`, where `nodes` contains the document chunks created by one of LlamaIndex's chunking methods.

One of LlamaIndex's main advantages is that chunk metadata is automatically created and stored within the `VectorStoreIndex` object. In contrast, LangChain may require manual metadata setup. Additionally, due to LangChain's modular design, the methods for handling metadata can vary depending on the backend vector store. However, LangChain's vector store integrations are not wrapped by a single class, offering more flexibility and granular control by exposing unique features of each vector store type.