AMERICAN UNIVERSITY OF BEIRUT

Fall 2025

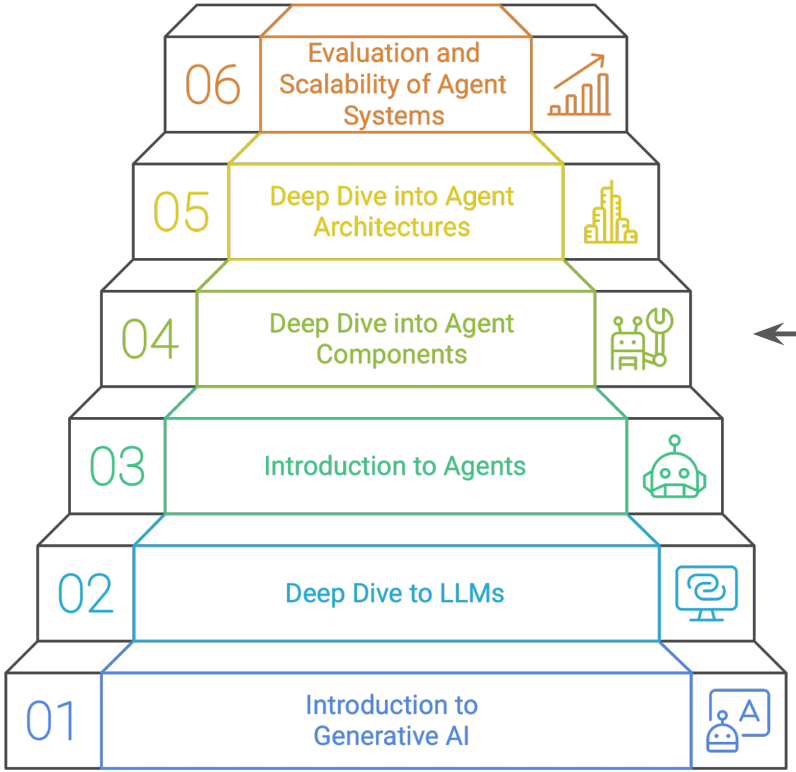
EECE 503P/798S: Agentic Systems

Chapter 6: Agent Memory

Lesson Objectives

- Understand different **types of agent memory**
- Learn and apply different **Retrieval Augmented Generation methods**
  - Understand what **Semantic Similarity** is, and the different methods of applying it
  - Understand what **vector databases** are, and how to use them
- Implement supplying different **types of external information to an agent** (graph, text, images, documents)

Course Outline



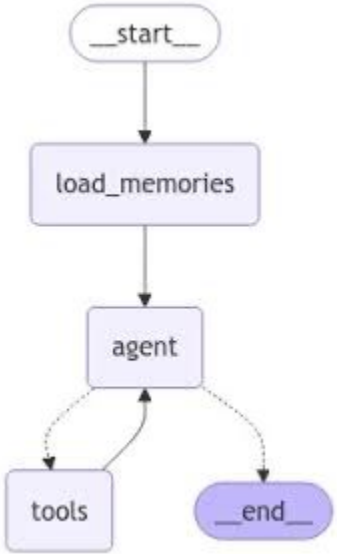
AMERICAN UNIVERSITY OF BEIRUT

Introduction

- Imagine talking to a friend who forgets everything you've ever said. Every conversation starts from zero.
- No memory, no context, no progress. It would feel awkward, exhausting, and impersonal.
- Unfortunately, that's exactly how LM behave today. They're full of information, yes, but they lack something crucial: **memory (context)**.

The need for context

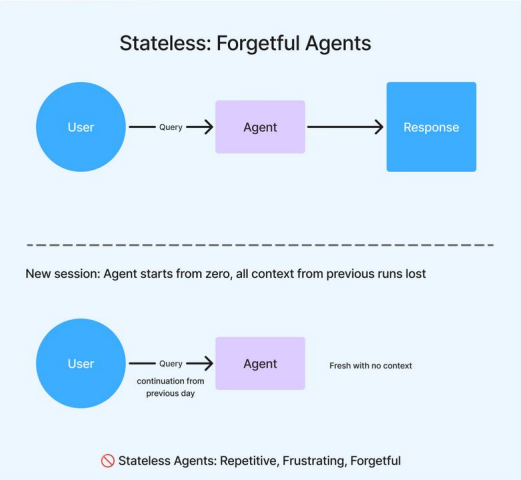
- Modern autonomous agents built on large language models (LLMs) do not inherently **“remember” past interactions** beyond their prompt context.
- By default, they are **stateless**, once a conversation or task ends, all **context** is lost unless re-provided.

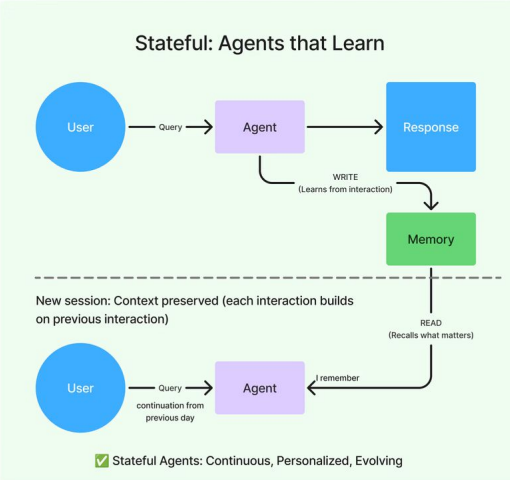


To move from stateless tools to truly intelligent, autonomous (stateful) agents, we need to give them memory, not just bigger prompts.

What is Agent Memory?

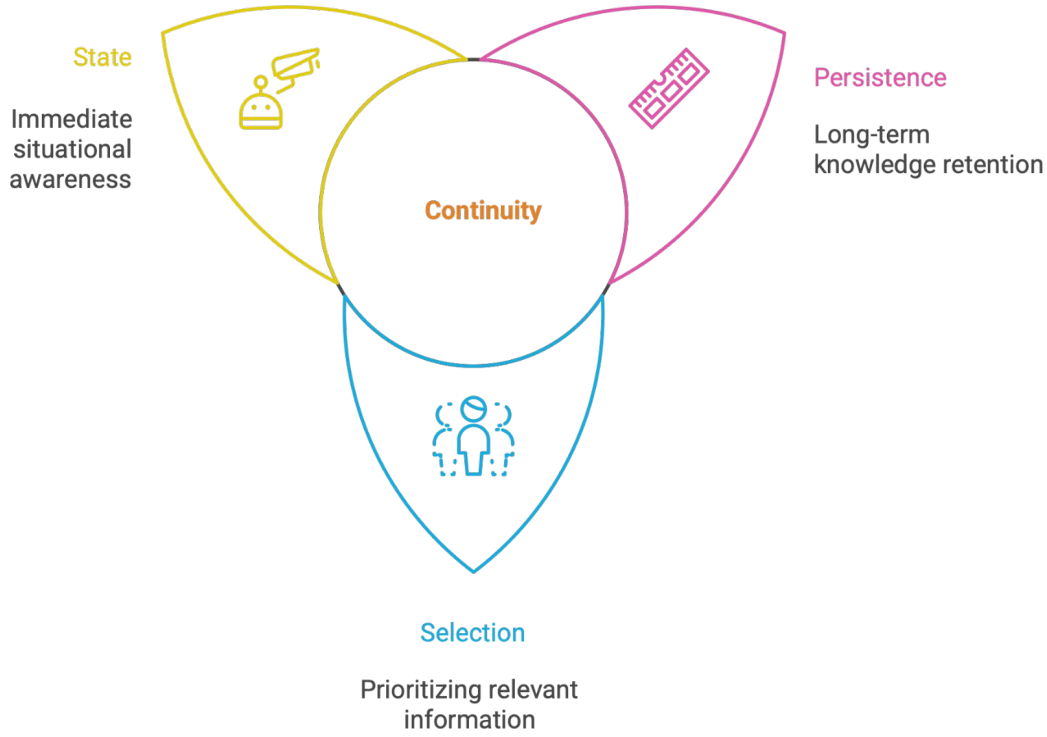
- **Memory** is the ability to retain and recall relevant information **across time, tasks, and multiple user interactions**.
- It allows **agents to remember** what happened in the past and use that information to improve behavior in the future.



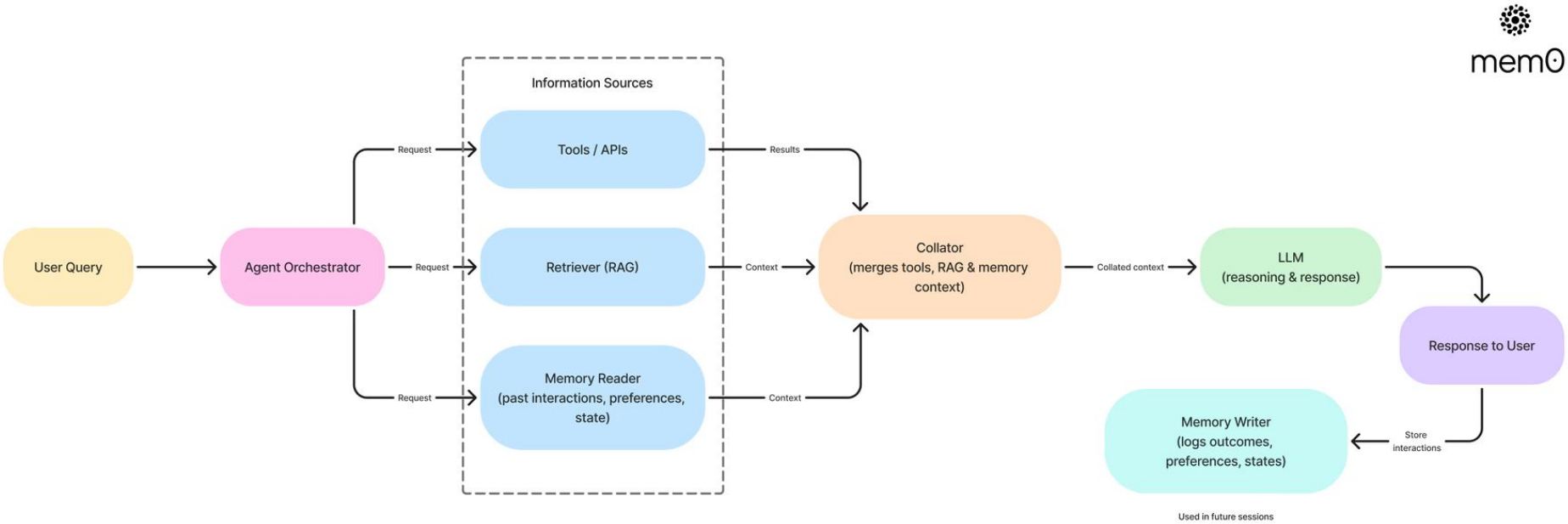


What is Agent Memory?

- Memory is about building a **persistent internal state** that **evolves** and **informs** every interaction the agent has, **even weeks or months** apart.
- Three pillars define memory in agents:



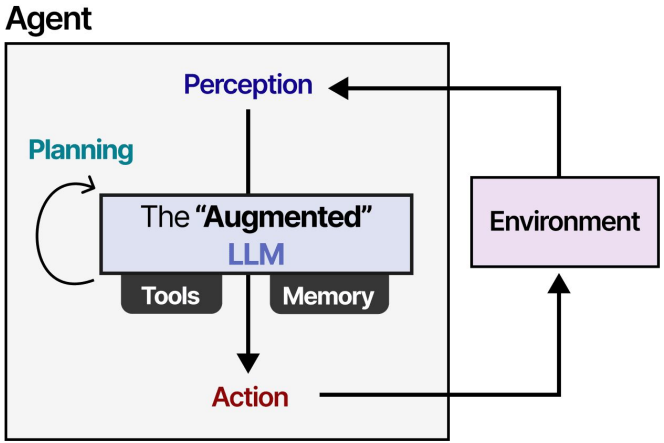
How does memory fit in the pipeline?



New models have context windows that can fit in hundreds of pages of text at a time. Doesn’t this solve the memory issue? No, it does not.

Memory Vs Long Context

- This goes far beyond the illusion of memory provided by a large context window.
- **Even with 100K-token contexts**, an LLM is **only carrying over data within a single session and still lacks true long-term understanding.**



Memory Vs Long Context

Long Context

Feeds the agent:

- All past user interactions
- All of the past tool calls and their output
- All system and capability instructions

Vs

Memory

Feeds the Agent:

- A comprehensive yet concise summary of most important past events, user preferences, and task considerations.

Memory Vs Long Context

Long Context

Consequences:

- Agent has to comprehend many different data structures at once
- Agent has to decide which part of the context is relevant for current task
- Agent has less reasoning tokens to perform the task since it also has to process all the context

Vs

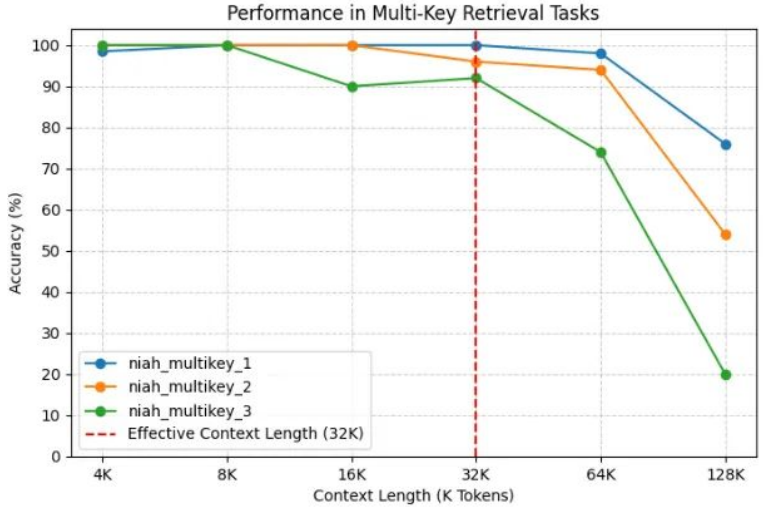
Memory

Consequences:

- Agent can focus on the next step
- Does not use up too many tokens, leaving room for image or document attachments for multimodal agents
- As the available context gets larger, the chain of events remains clear and simple

Dangers of Using Long Context

- The longer the context, the dumber the LLM.
- Larger context windows **require more GPU power** and **can lead to hallucinations** and slow responses.



Test conducted on Garnite 3.1- 8B model

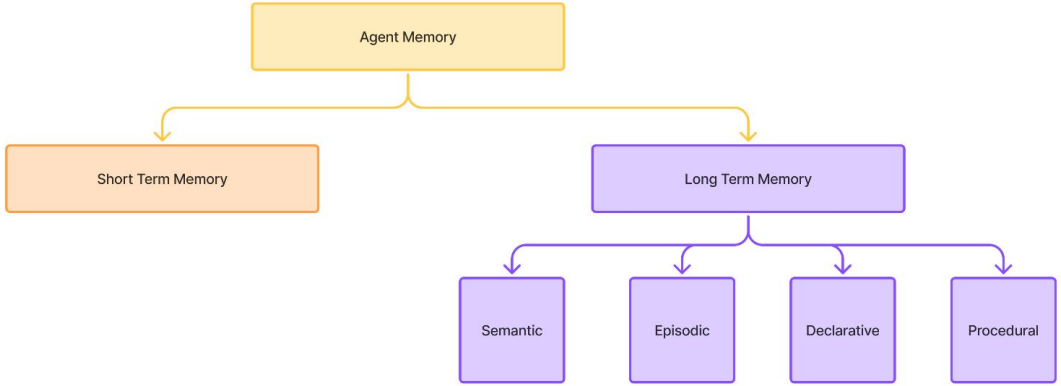
As the agent becomes more sophisticated, it has more information to manage and filter through.

Should we always feed a long summary of events into the content?

Types of Memory

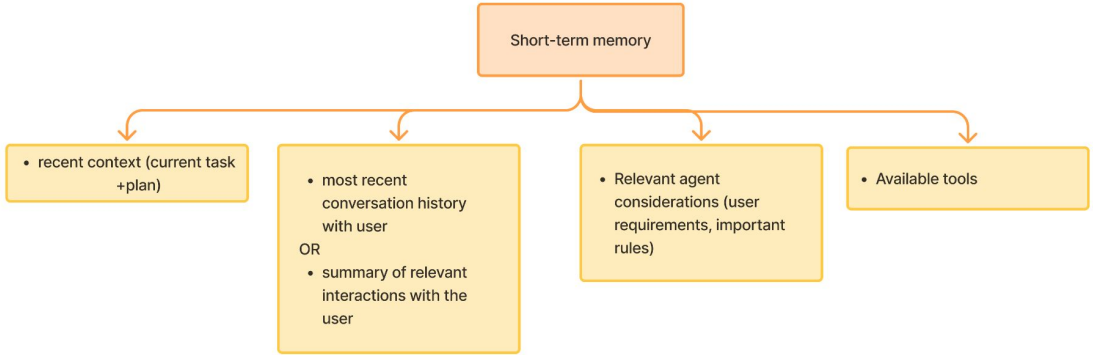
Types of Memory

- Key types include **short-term (working) memory** and **long-term memory**, with the latter subdivided into **episodic**, **semantic**, **declarative**, and **procedural** forms.

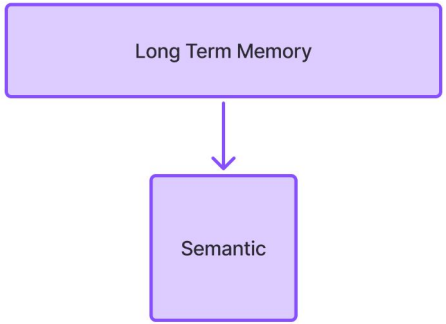


Short-Term Memory

- **Short-Term Memory (Working Memory):** is analogous to a conversation thread or a computer's RAM, once the session or "thread" ends, the data is gone unless saved.
- Short-term memory ensures ***local coherence***, letting the agent remember what was just said or done so it can respond fluently within a **single session**



Long-Term Memory - Semantic



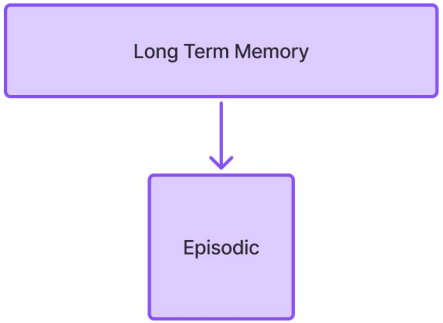
Definition

Semantic memory holds general facts, concepts, and world knowledge- the what the agent knows. The agent can explicitly recall and use this type of information.

Example

An agent should always remember that the user prefers answers in markdown.

Long-Term Memory - Episodic



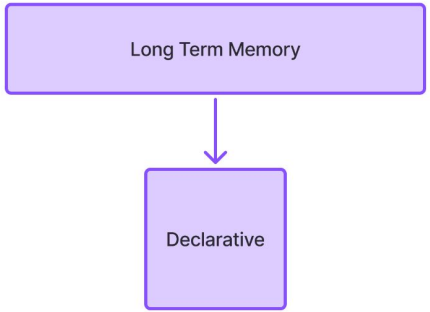
Definition

Episodic memory is memory of specific past events or experiences – the when/where of the agent's past actions.

Example

Episodic memory allows personalization and continuity: the agent can refer back to earlier dialogues or outcomes (e.g. "As I told you yesterday…") to maintain consistency.

Long-Term Memory - Declarative



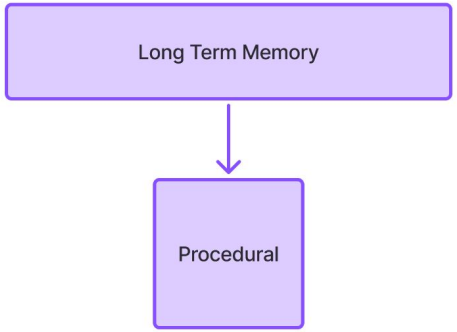
Definition

As noted, declarative memory refers to explicit knowledge that can be stated – encompassing both semantic and episodic memory. Some literature also uses declarative vs. non-declarative to contrast knowledge (facts/events) vs. skills.

Example

In practice, when building agents, we usually focus on implementing the semantic and episodic aspects (facts the agent should know, and records of what happened) and ensure the agent can retrieve these as needed.

Long-Term Memory - Procedural



Definition

Procedural memory encodes skills, procedures, or how-to knowledge – the know-how that guides actions

Example

For instance, an agent's core reasoning method (say, a ReAct strategy or a tool-use policy) as well as available tools lives in this procedural domain.

Long-Term Memory: Different Use Cases

- Although there is a clear distinction between short and long term memories, the **need to use both depends on the amount of context** or information the agent has access to.
- If you are building a simple **online search agent** -> **You only need a short term memory**
- If you are building a **deep research agent** -> You need a **long term and short term memory**

Hands on: 1-Short_Term_and_Long_Term_Memory.ipynb

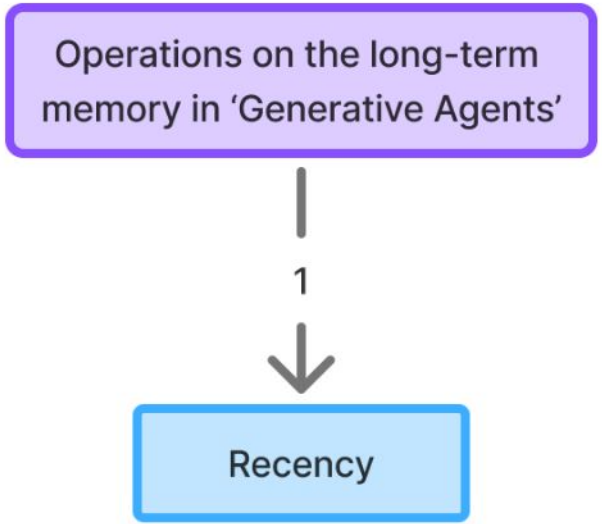
Applications of Agent Memory

Agents are relatively new, so what we define as memory is still somewhat fluid.

Earlier Applications of Agent Memory

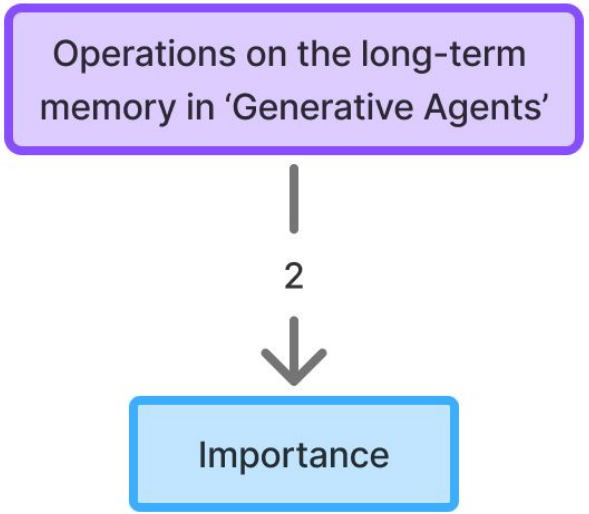
- **Generative AI Agents**: A research project by DeepMind and Stanford researchers that determined to test the limits of autonomous agents in **an open setting like a village**.
- They used a unique formula for filtering memory.

Earlier Applications of Agent Memory



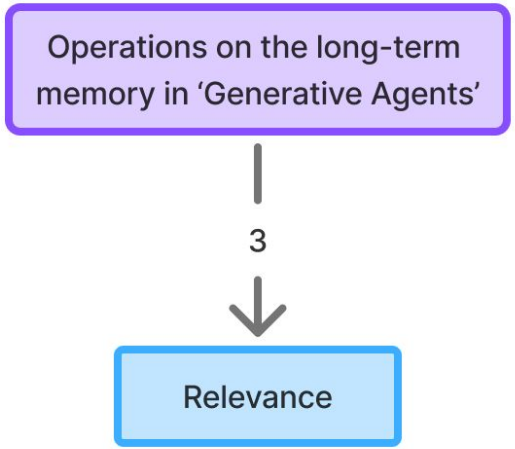
Assigns a higher score to events that happened most recently. It is implemented as an exponential decay function with a decay factor of 0.995

Earlier Applications of Agent Memory



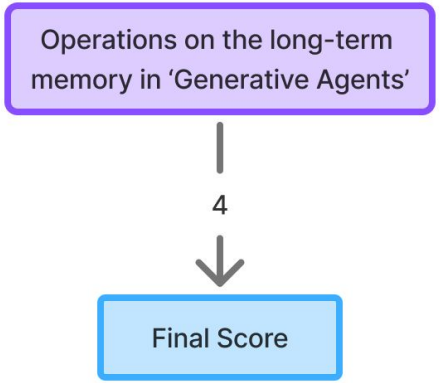
Assigns an int value between 1 and 10 to events with high importance and effect. Added through asking an LLM to assign this number.

Earlier Applications of Agent Memory



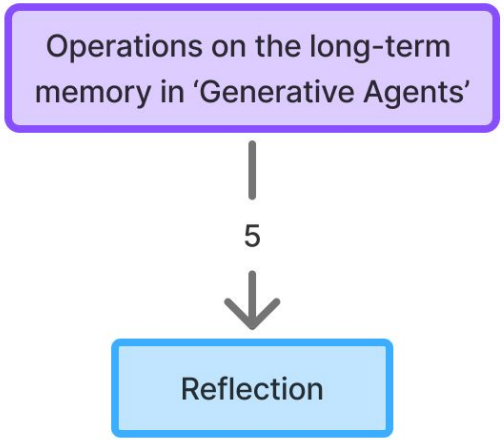
Assigns a higher score to memory objects that are related to the current situation. All the memories are embedded and a cosine similarity between the current perception/question and the memories is calculated to attain the relevance.

Earlier Applications of Agent Memory



To calculate the final retrieval score, they normalize the recency, relevance, and importance scores to the range of [0, 1] using minmax scaling.
The retrieval function scores all memories as a weighted combination of the three elements: $score = \alpha_{recency} \cdot recency + \alpha_{importance} \cdot importance + \alpha_{relevance} \cdot relevance$. In the implementation, all αs are set to 1.
The top-ranked memories that fit within the language model’s context window are included in the prompt

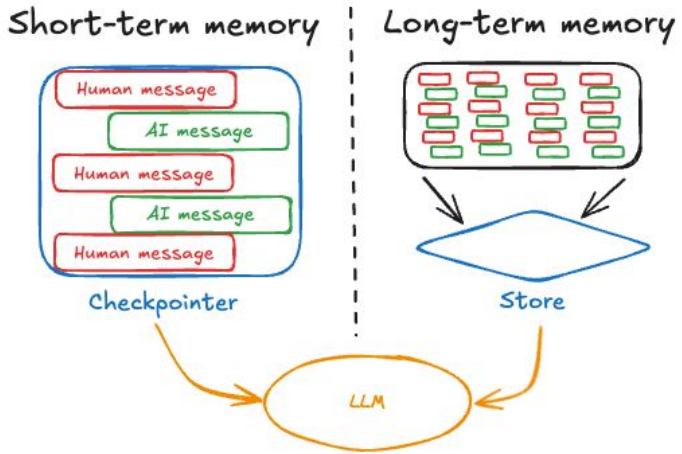
Earlier Applications of Agent Memory



A new type of memory is appended to the memory stream called a reflection. It is periodically calculated (in this case every 150 observations) and is generated by an LLM that is given the 150 observations and prompted to generate 5 high level insights on the observations, each insight having reference to the indexes of the observations used to derive it.

Earlier Applications of Agent Memory

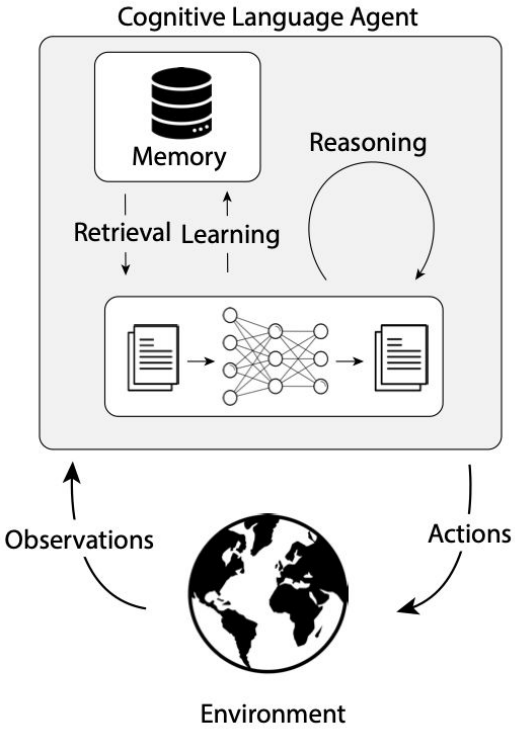
- Even though the AI community is converging towards using the definitions of short term and long term memory we used earlier, this is still a **fluid concept** where you can innovate as required depending on the **complexity and parameters** of your application.



Earlier Applications of Agent Memory

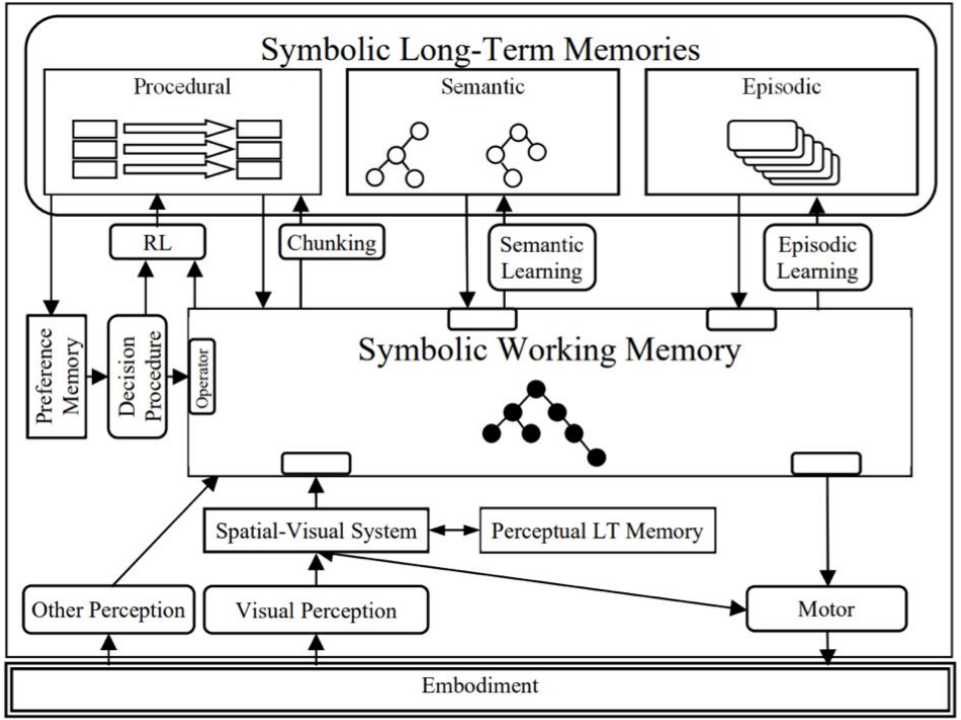
Another notable architecture is '**Cognitive Architectures for Language**'

1. New information can be **stored directly in long-term memory**: **facts can be written to semantic memory**, while **experiences** can be written to **episodic memory.**
2. This information can later be retrieved back into **working memory** when needed for decision-making.
3. Soar is also capable of writing new productions into its **procedural memory**, effectively updating its source code.



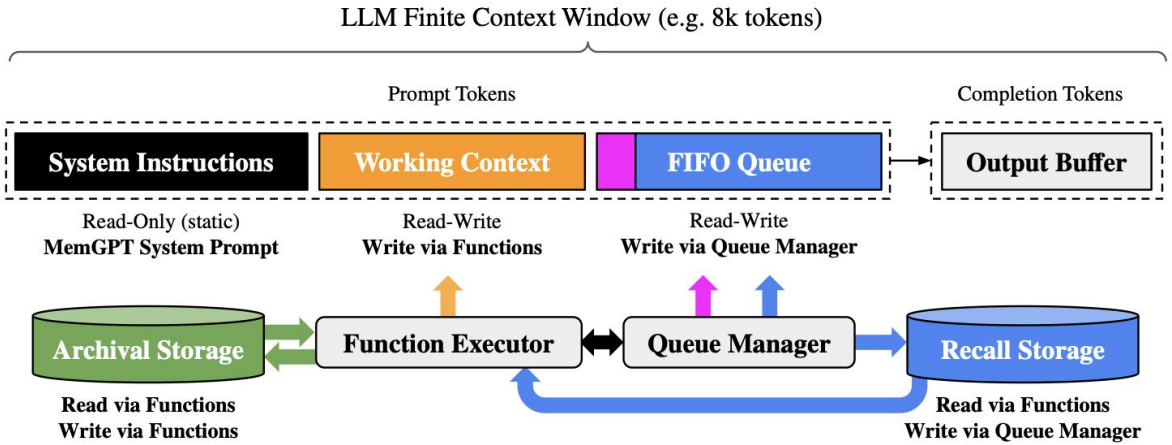
Earlier Applications of Agent Memory

‘Cognitive Architectures for Language’ memory illustrated.



Earlier Applications of memory

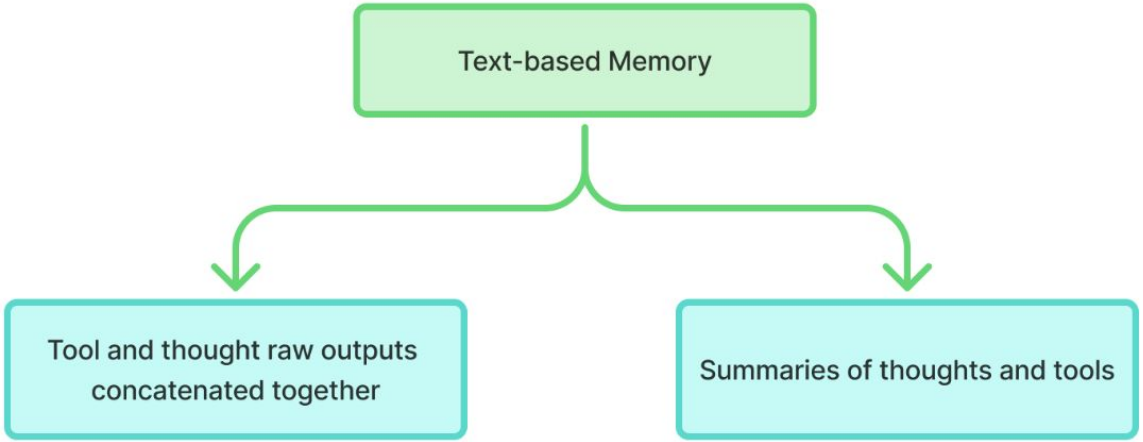
- In MemGPT, a **fixed-context LLM processor** is augmented with a hierarchical memory system and functions that let it manage its own memory.
- The LLM's **prompt tokens (inputs)**, or main context, consist of the **system instructions, working context,** and **a FIFO (First in- First out) queue**.



Memory Storage Types

Is the memory always stored as natural text? What other forms can we store the memory in?

Text-Based Memory



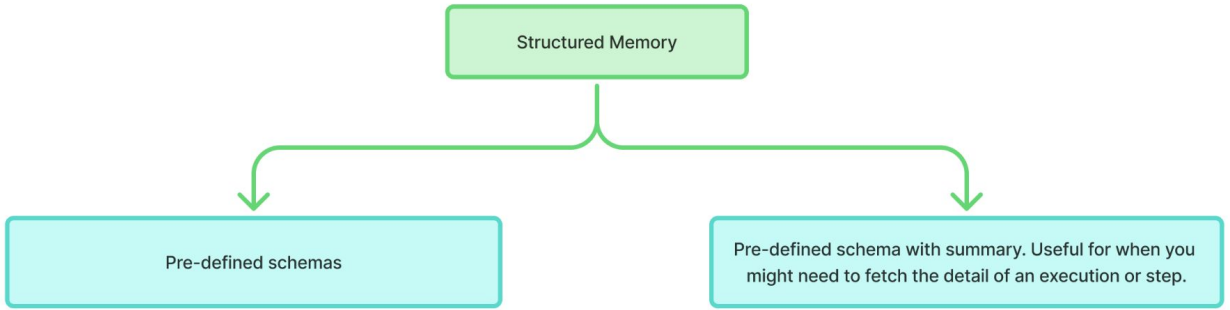
Example:

"I want to look up the year apple was founded" + "Online Search Result: Apple was founded in 1976" + "Next I need to look up what year the Macintosh was first introduced" + ...

Example:

"The next step was to verify the year which apple was founded, so we used the online tool that returned the year 1976"

Structured Memory



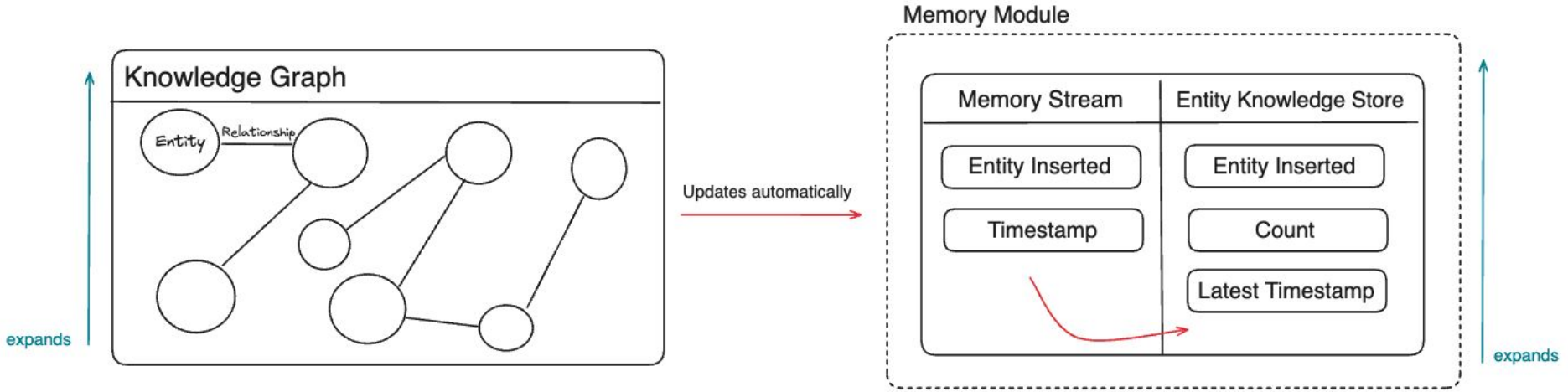
Example:

```
agent_memory = [{
    'type': 'Thought',
    'outcome': 'Decided to look up the founding year of the company Apple'
  },
  {
    'type': 'Tool_Call',
    'outcome': 'Apple was founded in 1976'
  }
]
```

Example:

```
agent_memory = {
  'agent_memory_summary': '..',
  'agent_logs':
  [{
    'type': 'Thought',
    'outcome': 'Decided to look up the founding year of the company Apple'
  },
  {
    'type': 'Tool_Call',
    'outcome': 'Apple was founded in 1976'
  }
]}
```

Graph Memory



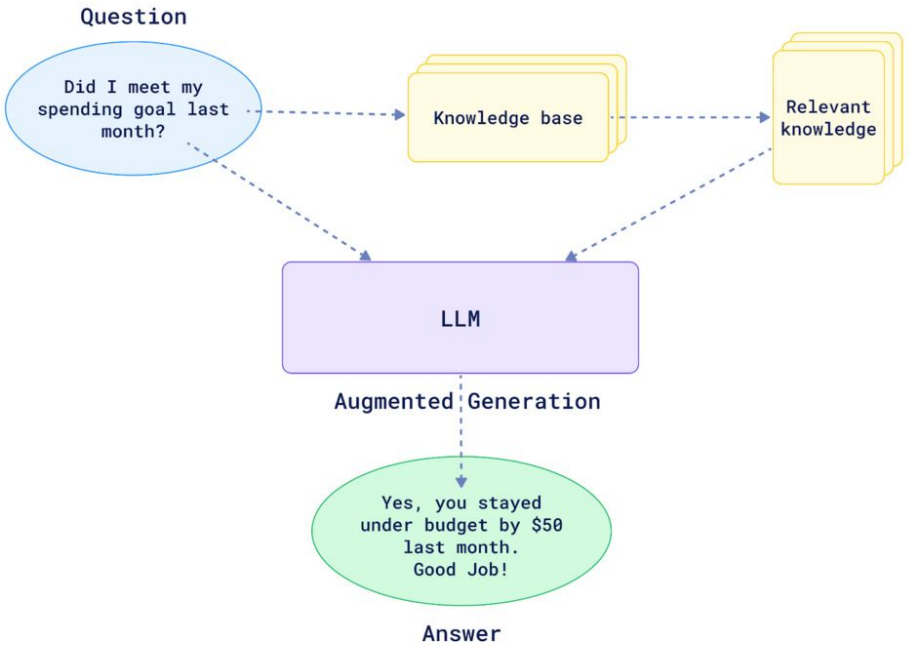
AMERICAN UNIVERSITY OF BEIRUT

RAG

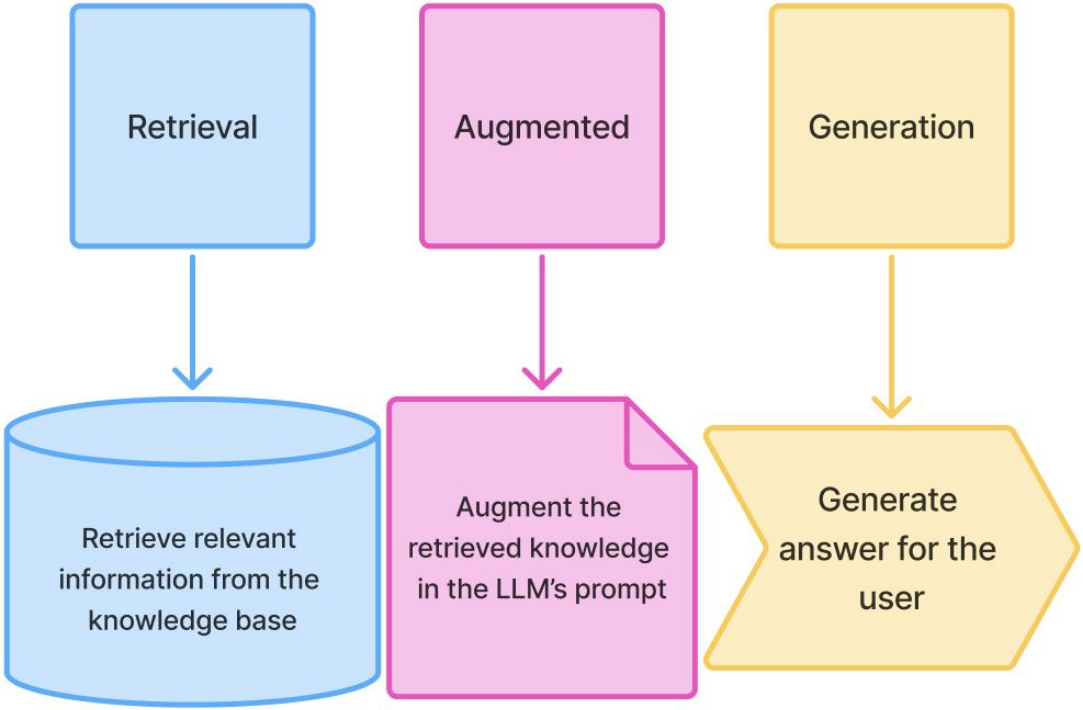
In ‘Generative Agents’, we saw how they used a formula to decide which memory is appended to a prompt. What other methods of retrieval are there?

Retrieval Augmented Generation

- Retrieval augmented generation is a method that allows an LLM **access to a knowledge base**.
- Before generating a response, the system fetches **relevant documents or facts** and **injects them into the prompt**.

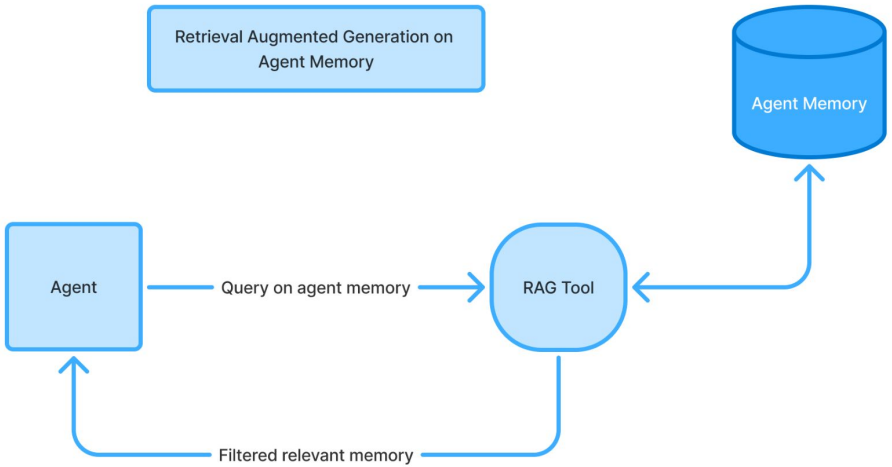


Retrieval Augmented Generation



Retrieval Augmented Generation

- Retrieval Augmented Generation was not initially proposed for Agent Memorya.
- Rather, it was to **supply the LLM** with any **relevant information** from an **external knowledge source**.
- However, this is one of the very effective methods of **filtering** out the **agent's memory** to extract what is relevant.



Retrieval Augmented Generation

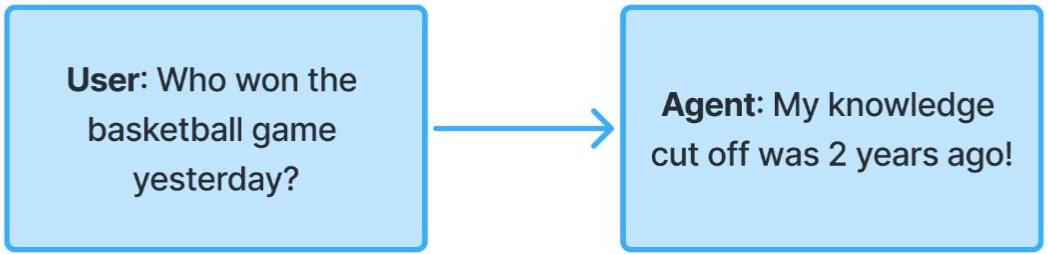
Aspect	RAG: Retrieval-Augmented Generation	Memory in Agents
Temporal Awareness	No concept of time or sequence	Tracks order, timing, and evolution of interactions
Statefulness	Stateless; each query is independent	Stateful; context accumulates across sessions
User Modeling	Task-bound; agnostic to user identity	Learns and evolves with the user
Adaptability	Cannot learn from past interactions	Adapts based on what worked or failed

What is the point of having RAG? Aren't LLMs knowledgeable enough? Aren't LLMs smart enough to handle long context?

RAG- Why do we need it?

Problem 1: **Hallucinations and Outdated Knowledge**

Even though LLMs like GPT-4 have **billions of parameters** trained on vast data, enabling tasks like translation, summarization, and Q&A, they face key challenges: **outdated knowledge, hallucinations**, and **lack of source citations.**

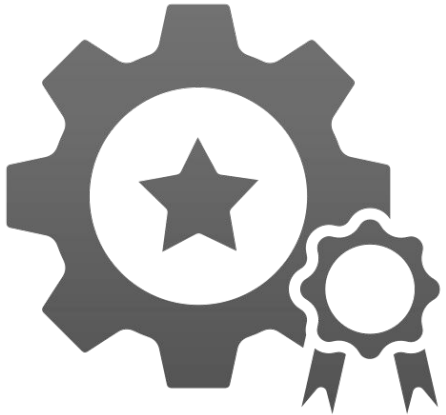


RAG- Why do we need it?

Problem 2: Hard to **specialize**

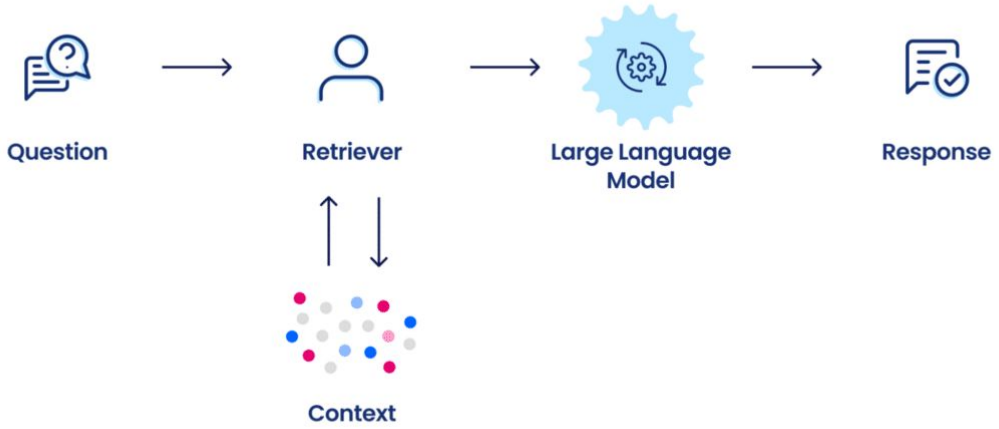
While we can instruct an LLM to behave like a useful customer service agent, it:

1. **Cannot ingest your entire customer service manual** for every call (costly, large latency, simpler answers).
2. Is not feasible to **train** (on your customer service Q&A) and **host** your own LLM (less conversational, might hallucinate, would need to retrain for every change in the manual)



RAG- Why do we need it?

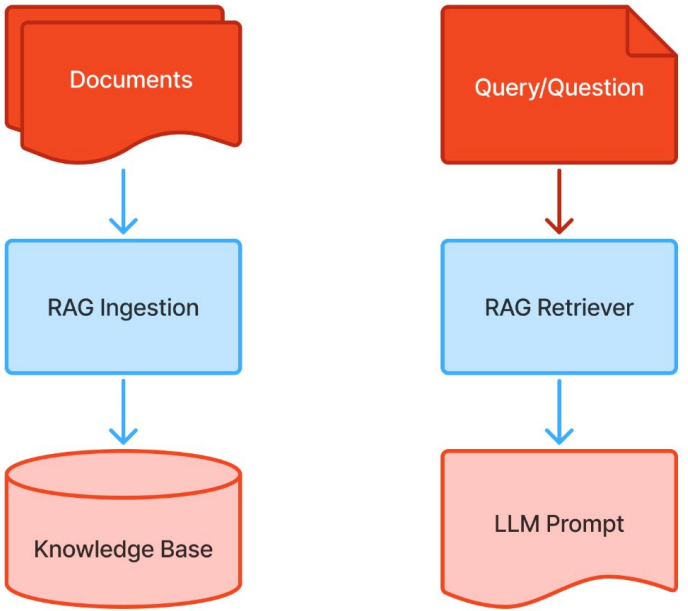
- When the **agent has access to a lot of information**, like user documents, long-term memory, and conversation history, we need an efficient way of retrieving any piece of information
- **Solution**: RAG. Whenever the agent has a question to be answered by the agent's documents or memory, we can **query the knowledge base** to get an answer.



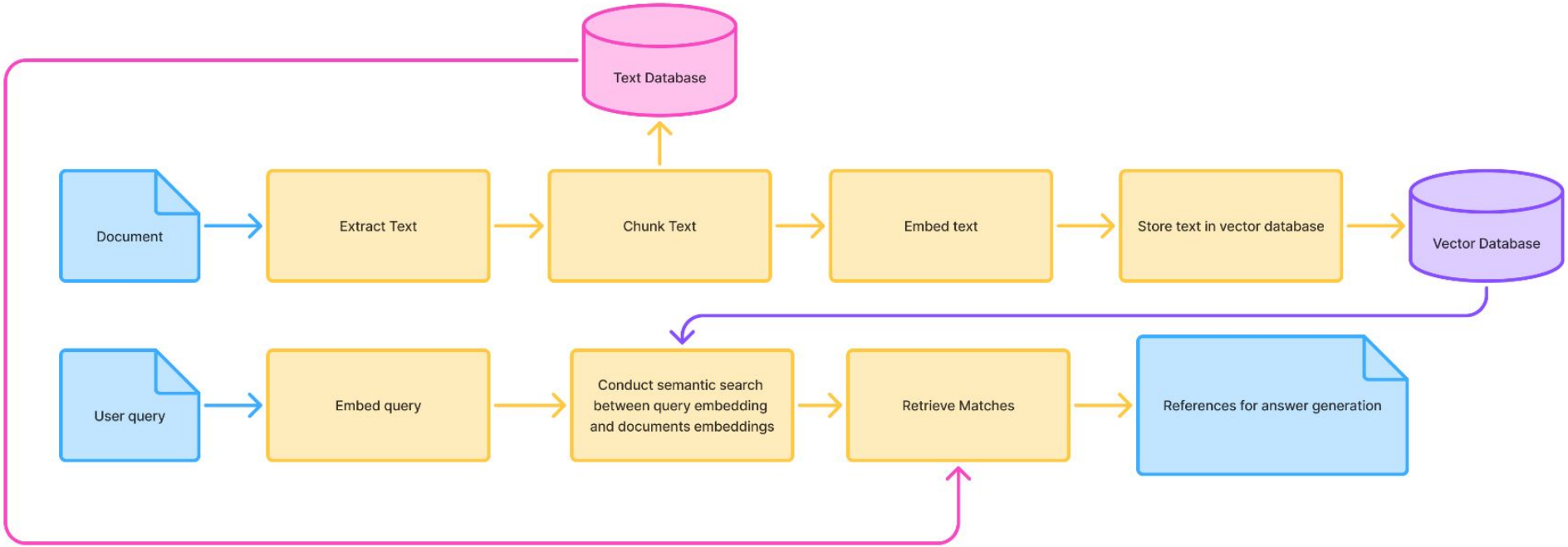
RAG- The basic pipeline

RAG has two main pipelines:

1. **Information Ingestion**: processing documents/text/… to be saved in the knowledge base
2. **Retrieval**: retrieving information from the knowledge base to to augment in the prompt.



RAG- The basic pipeline



Hands on: 2_RAG_Pipeline.ipynb

Let’s zoom in to RAG ingestion pipeline

RAG Document Ingestion Pipeline

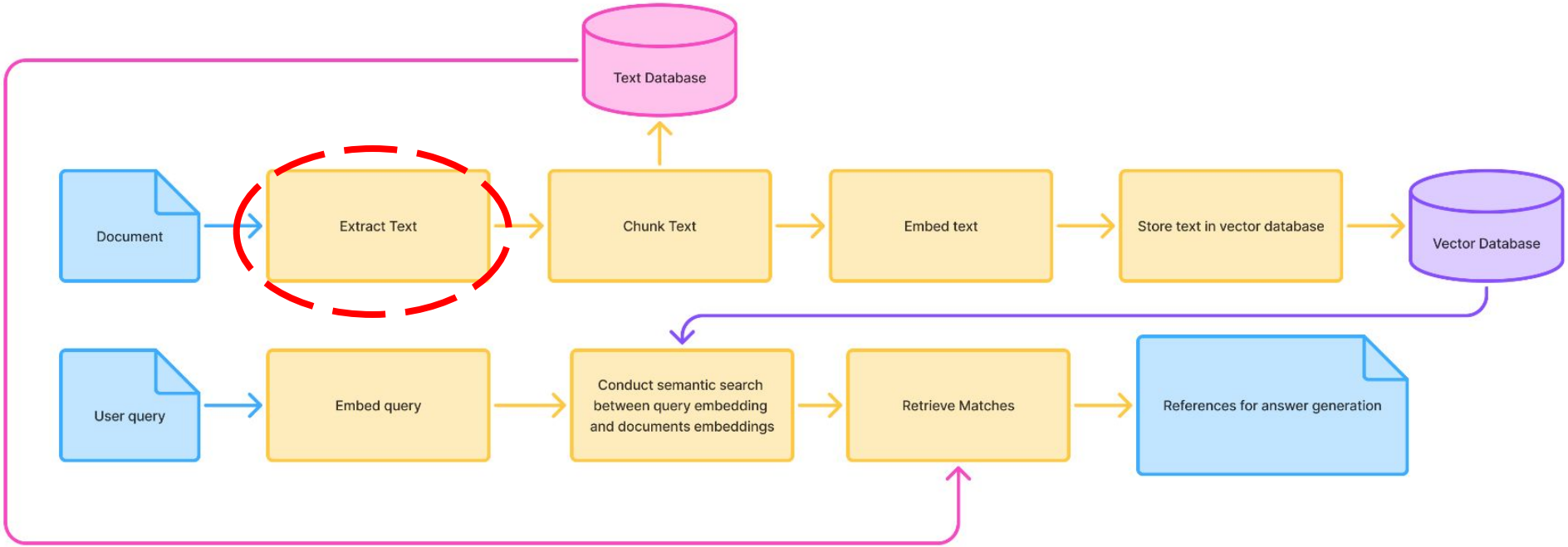
Let’s assume that the user uploaded a PDF.

How do we add it to the knowledge base?

What does the knowledge base look like?

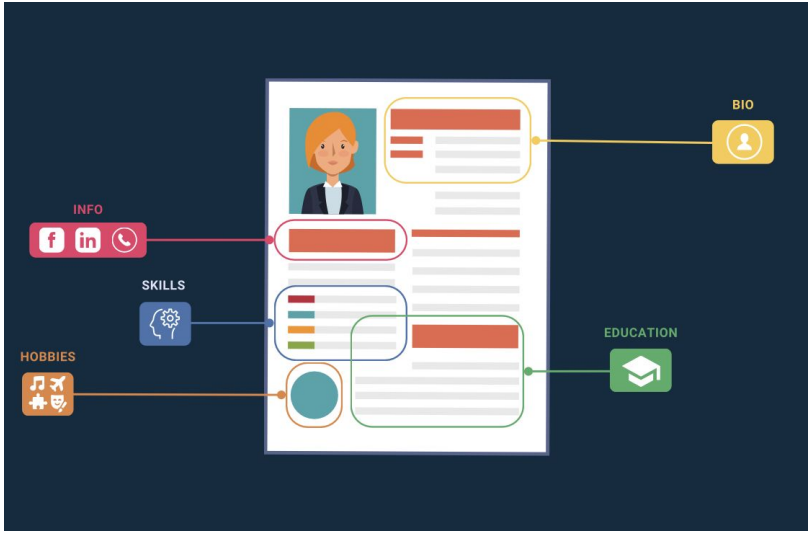
Extracting text from documents

RAG- The basic pipeline



Document Parsing

- Parsing isn't just about **extracting** text.
- It's about **preserving** structure, context, and relationships within the data.
- Get this wrong, and your **entire RAG pipeline suffers**.

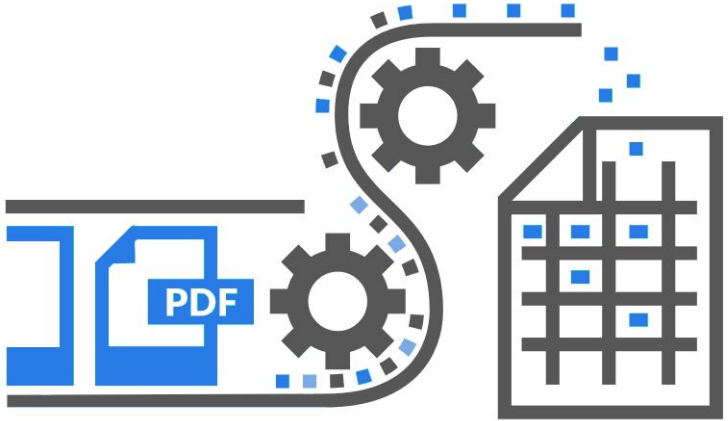


Document Parsing

Document parsing is the process of **recognizing/examining data** in a document and extracting it.

There are **multiple approaches** to document parsing:

1. Extracting text from document metadata
2. Optical Character Recognition
3. Using a Large Language Model



Document Parsing - Using Metadata

- There are **libraries** that open PDF files, typically in binary read mode, and load its content into memory or a suitable object structure.
- This allows the library to access the PDF's **internal components** and consequently **extract the text elements**.

```python
import pymupdf

doc = pymupdf.open("a.pdf") # open a document
out = open("output.txt", "wb") # create a text output
for page in doc: # iterate the document pages
    text = page.get_text().encode("utf8") # get plain text (is in UTF-8)
    out.write(text) # write text of page
    out.write(bytes((12,))) # write page delimiter (form feed 0x0C)
out.close()
```

```python
from PyPDF2 import PdfReader

reader = PdfReader("example.pdf")
number_of_pages = len(reader.pages)
page = reader.pages[0]
text = page.extract_text()
```

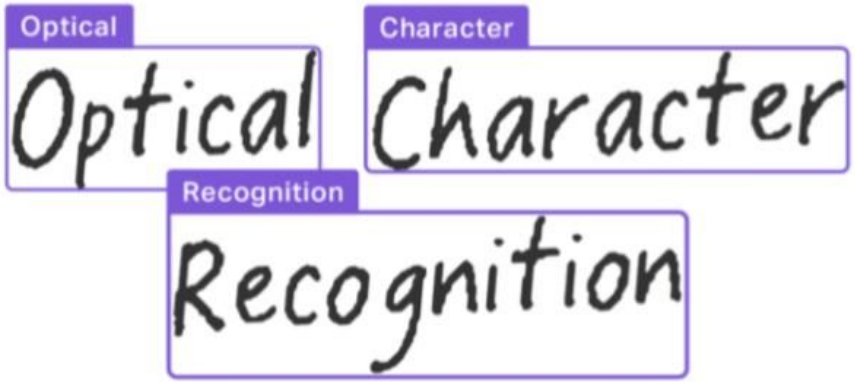
Document Parsing - Using Metadata

While algorithmic text parsing of PDFs is a **neat** and **fast approach**, it isn't without its limitations.

1. The parsed text would need **cleaning**. It would be full of '\n' and other formatting delimiters.
2. Tables and multi-column formats are **not parsed** in a readable format that preserves the layout.
3. **Images are ignored.**

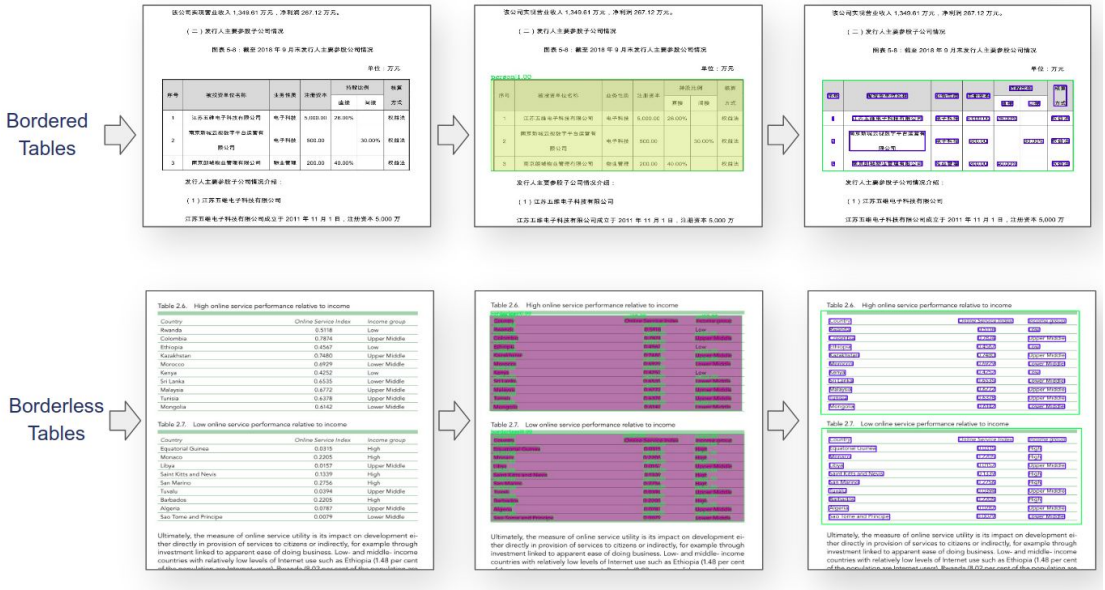
Document Parsing - OCR

- Optical Character Recognition (OCR) is technology that **converts images** of text, whether typed, handwritten, or printed, into machine-readable, editable text data.



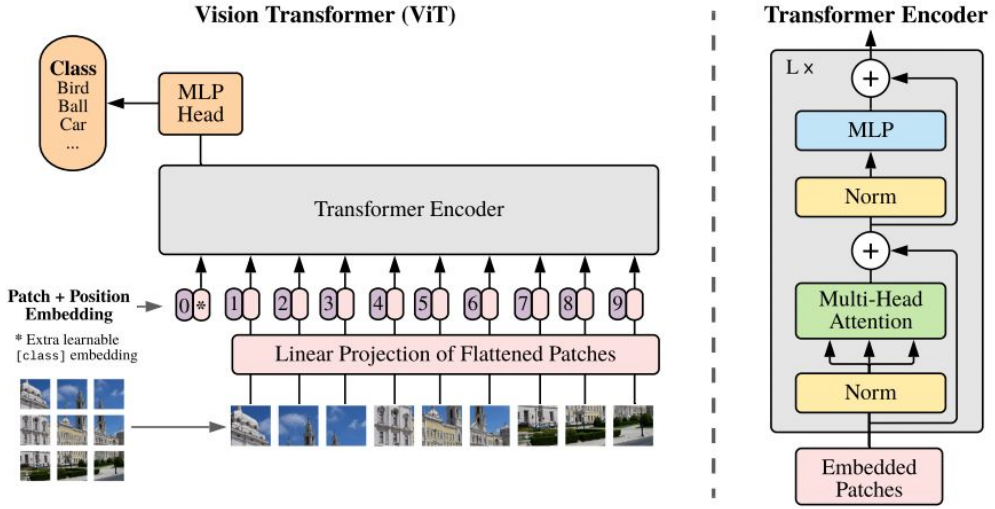
Document Parsing - OCR

- OCR can be used to **parse charts, tables,** and **multi-column data**.



Document Parsing - OCR Models

- OCR models do not follow one architecture.
- They could be **LSTMs, CNNs, VLMs, or RNNs**.

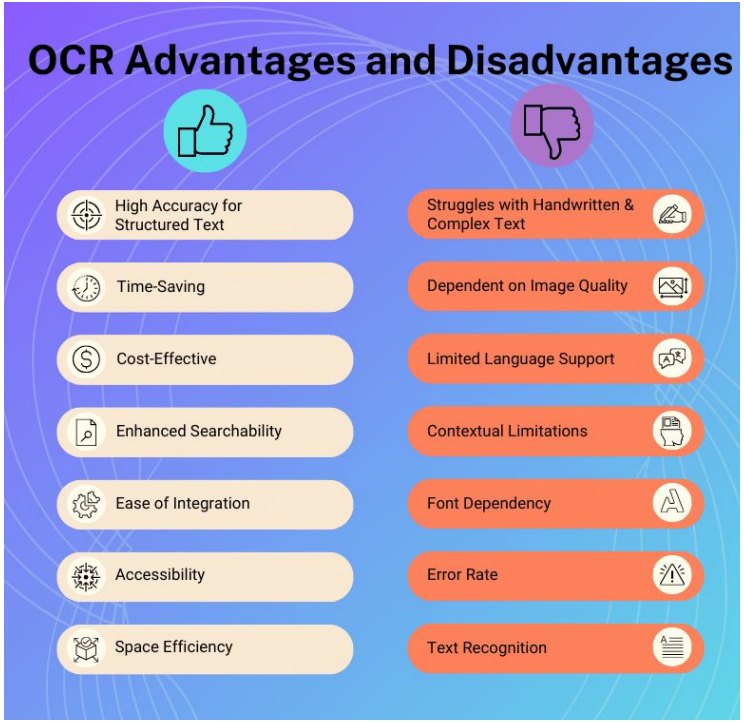


Document Parsing - OCR Models

Feature	GOT-OCR2	Tesseract OCR	UReader	LLaVA-NeXT
End-to-End Model	✅	❌	❌	✅
Multi-Language Support	✅	✅	✅	✅
Scene & Document OCR	✅	❌	✅	✅
Table & Math Formula OCR	✅	❌	✅	❌
Interactive OCR	✅	❌	❌	✅
Model Size	**580M**	-	1.2B+	7B+

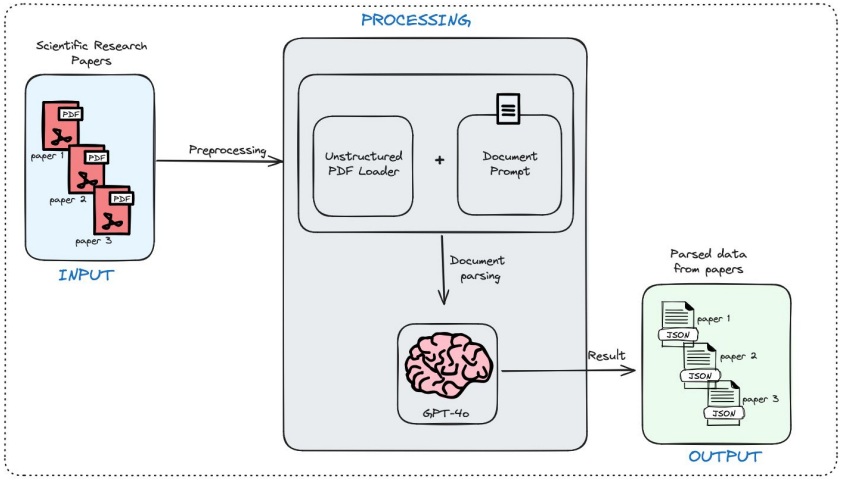
Document Parsing - OCR Shortcomings

- OCR models were made for **character recognition,** so they lack word, sentence, and context awareness.
- For example, if a title page had a lot of spacing, it might capture each letter in the title as individual words.
- Another **disadvantage** is that images are ignored as well.



Document Parsing - Using LLMs

- Multimodal LLMs, like Claude 3.7 Sonnet and Qwen, accept **different types** of documents as inputs along with the prompt.
- So, you can pass an **entire document** to the LLM, and instruct it to **extract all the text** inside the document.



DOCUMENT PARSING WORKFLOW WITH LLM (Author: Zoumana Keita)

Document Parsing - Using LLMs

- The output received by using LLMs yields a **more accurate result**- theoretically. However, this is not always the case.

Parsing Using LLMs

Advantages:

1. Specialized parsing based on instructions (**parse in markdown, parse in HTML, preserve structure,..**)
2. Layout-aware parsing: **parses any layout** without mixing the order of the text.
3. **Can describe images, charts, and tables** to the level of details decided by the user

Disadvantages:

1. **Very costly** as the number of tokens per page is dependent on the image size and not the number of words inside the image.
2. **Takes much longer to process** than other parsing methods since the input and output token numbers are large
3. The longer the input (larger the document), the **model is less likely to follow instructions.** Thus, it sometimes adds '...' instead of parsing the rest of the paragraph. It might also **describe charts at a very high level**- losing important numbers and statistics.

Document Parsing - Approach Tradeoffs

The approach you will take to parse your documents depends on the following factors:

1. **Your desired latency**: Do you care if parsing takes a long time as long as the result is accurate?
2. **Your document's complexity**: Do you want images described, data in charts listed, and complex layout handled?
3. **Your document's source**: Are you dealing with handwritten text, receipts, types PDFs

Document Parsing - Advanced Cases

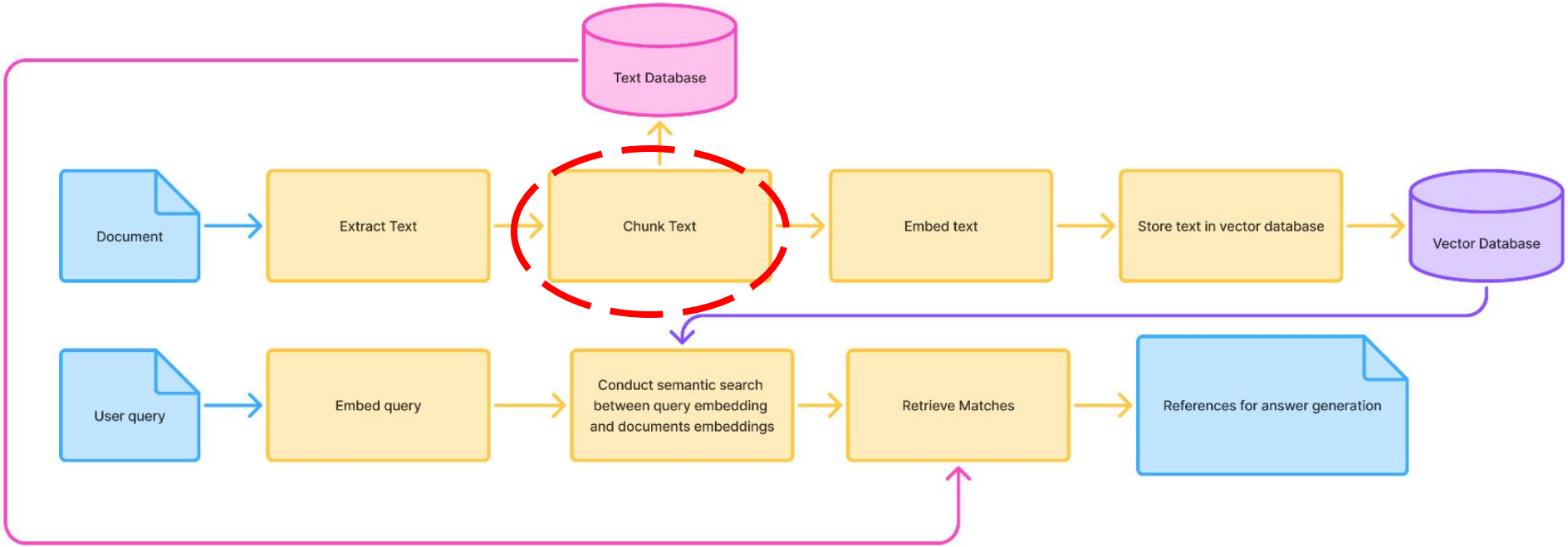
Some complex cases might require parsing techniques that LLMs, OCR, and algorithmic parsers **do not cover**. For example:

1. Identifying and extracting exact **locations of images, charts, and visualizations** to be referenced later on.
2. **Having 100% accuracy** on text parsing for any type of document. All methods described above have a margin of error depending on the document.

Hands on: 3_Document_Parsing.ipynb

Now that we have extracted text from the documents, we need to chunk them before embedding.

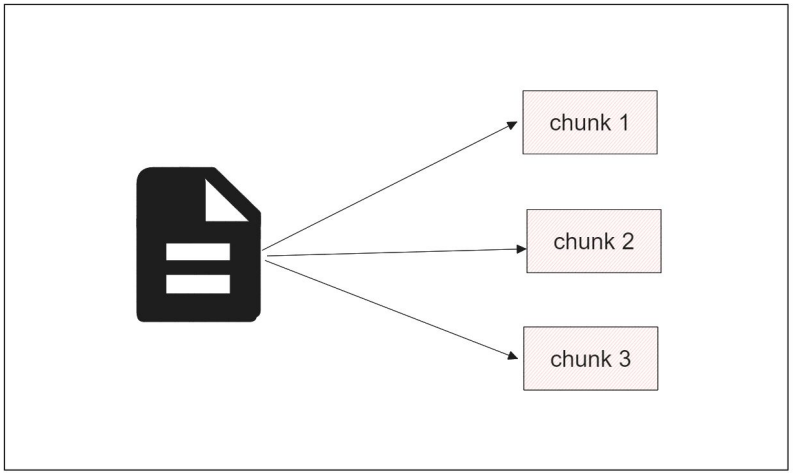
RAG- The basic pipeline



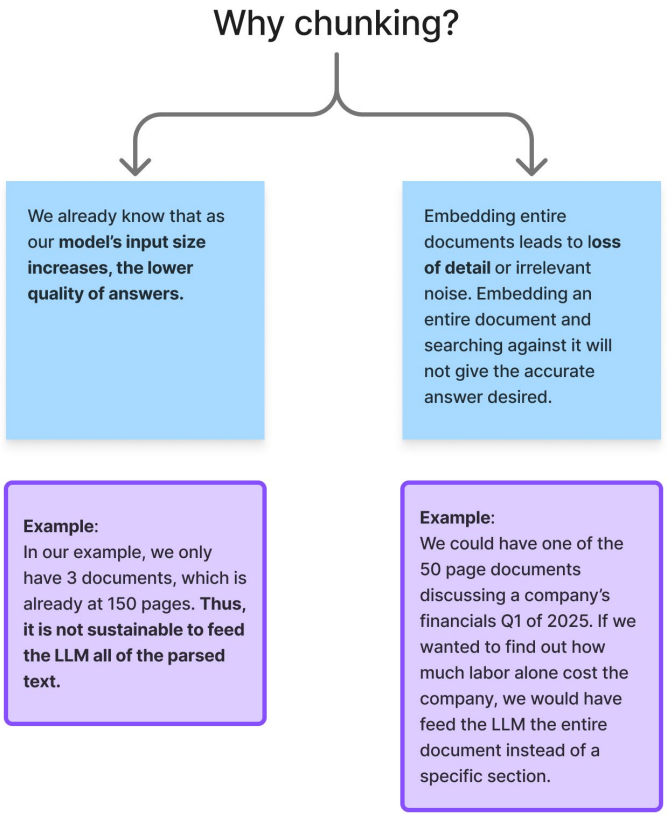
Why chunking?

Let’s say we parsed 3 documents, each 50 pages long.

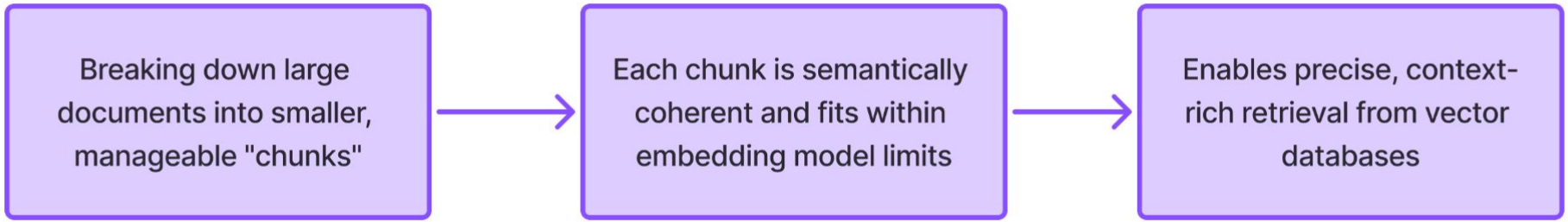
- Why do we need to **break down** the text into **smaller chunks** before embeddings?
- Isn’t the **context** of the LLM **large enough** to handle all the parsed text?
- Don’t we **lose context** when we dissect the text into smaller pieces?



Why Chunking?



What is chunking exactly?

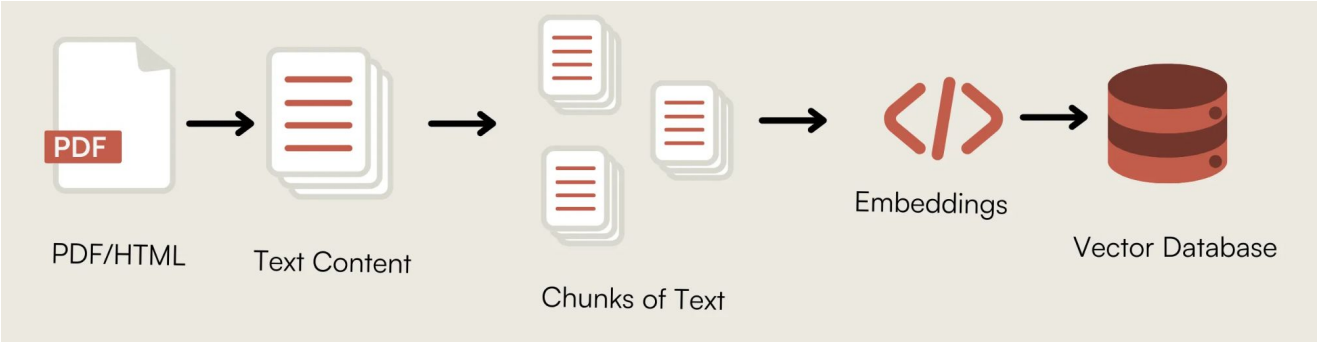


Choosing Chunk Size

Chunking isn't as straightforward as splitting the text into sentences or even paragraphs:

- **Too large**: chunks contain mixed topics, confuse retrieval, increase cost
- **Too small**: chunks lose context, reduce recall, fragment meaning

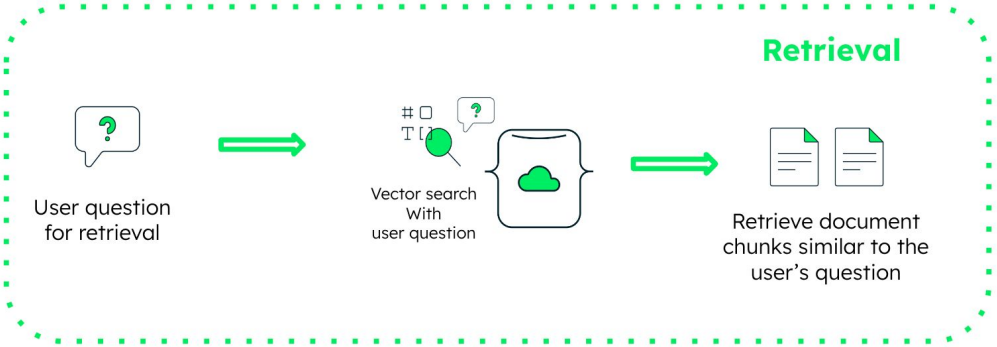
Optimal chunk size balances **context preservation** and **retrieval precision**



Choosing Chunk Size

"If chunk size wildly differs from query size, similarity scores drop" - Roie Schwaber-Cohen, Pinecone

- **Large chunks** dilute relevance; **small chunks** improve precision but may miss broader context
- **Metadata tagging** helps link chunks back to original documents for richer context



Choosing Chunk Size

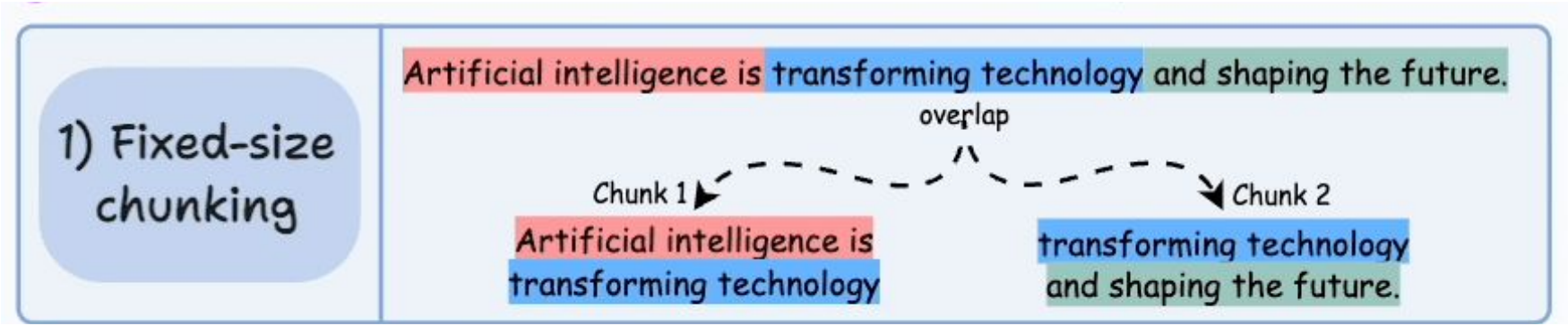
There's no single "rule of thumb" for RAG chunk size, but common starting points are **between 128–512 tokens for smaller chunks** and **512–1024 tokens for larger ones**, with **1024 tokens often cited as a good balance** for accuracy and efficiency.

The optimal size depends on factors like:

1. Document complexity,
2. Query type (fact-based vs. broad context),
3. LLM's context window, and performance goals

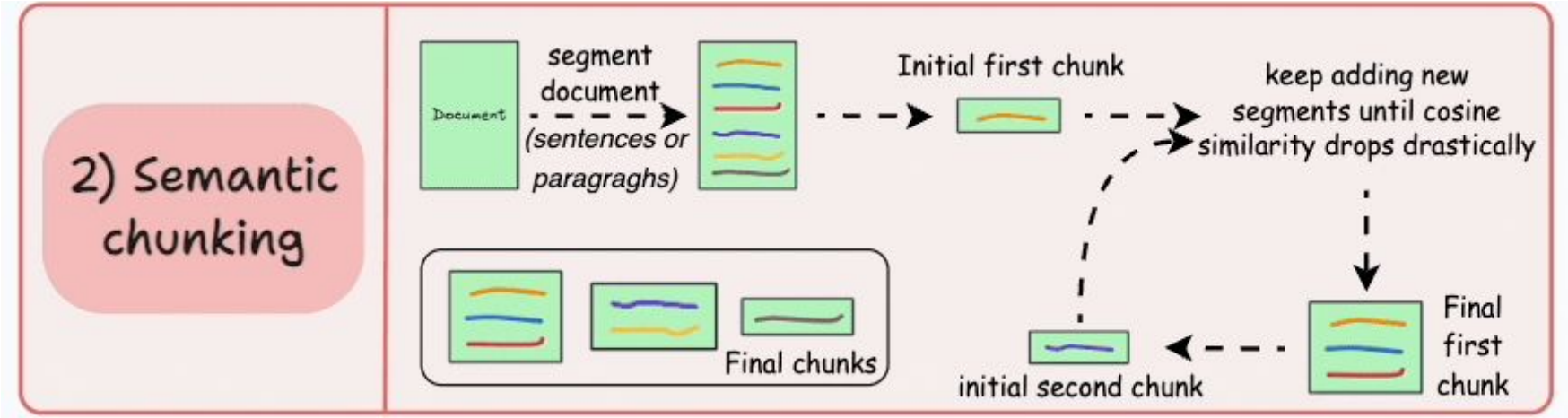
Chunking Strategies

1. Fixed-size chunking: This is the most crude and simplest method of segmenting the text. It breaks down the text into chunks of a specified number of characters, regardless of their content or structure.



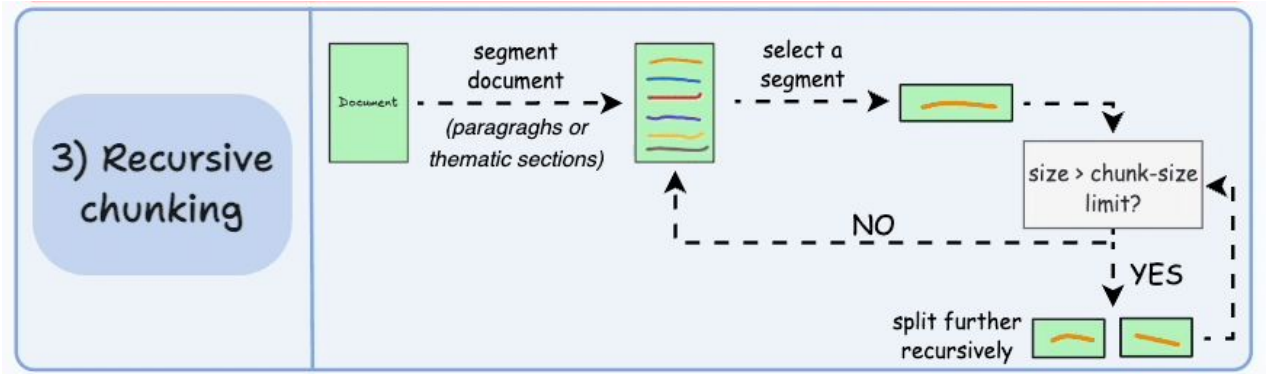
Chunking Strategies

2.Semantic chunking: This chunking method aims to extract semantic meaning from embeddings and then assess the semantic relationship between these chunks. The core idea is to keep together chunks that are semantic similar.



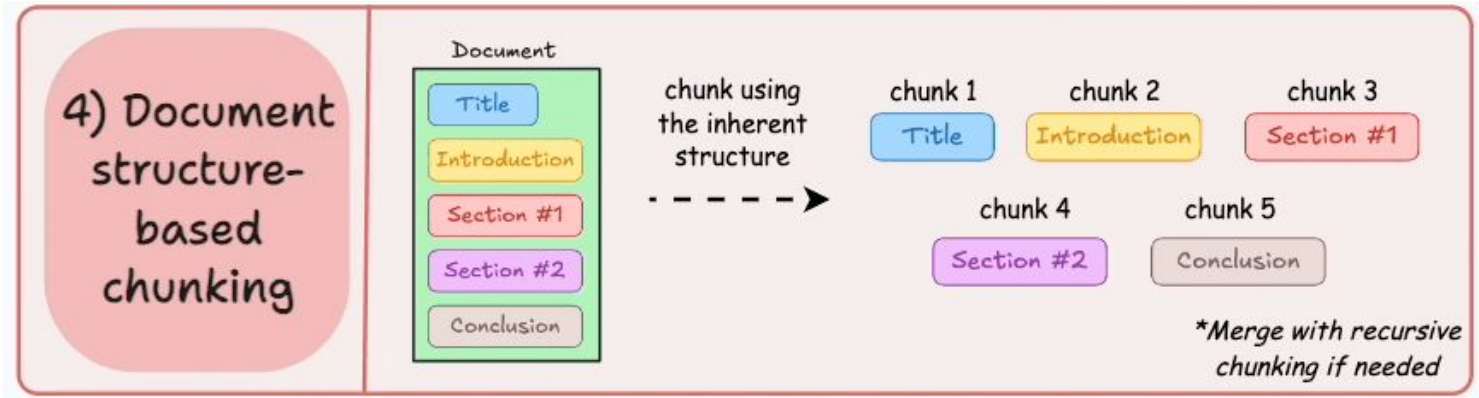
Chunking Strategies

3.Recursive Chunking: In this method, we divide the text into smaller chunk in a hierarchical and iterative manner using a set of separators. If the initial attempt at splitting the text doesn't produce chunks of the desired size, the method **recursively calls itself** on the resulting chunks with a different separator until the desired chunk size is achieved.



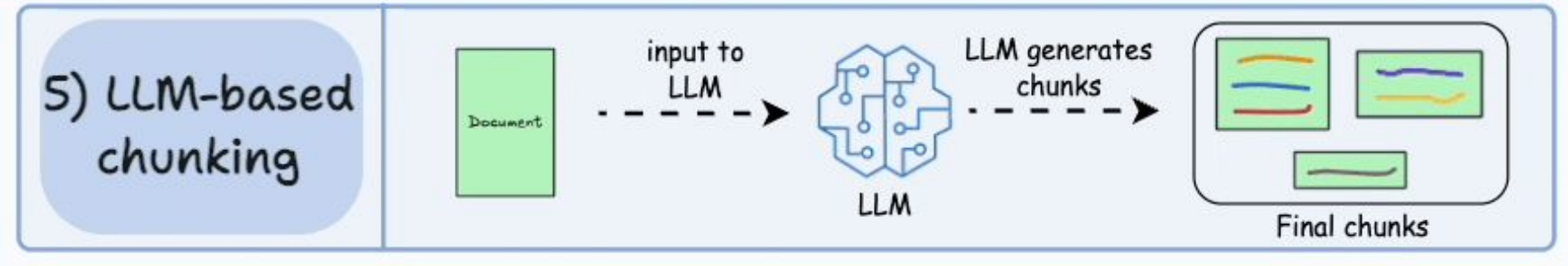
Chunking Strategies

4. Document Structure-based Chunking: In this chunking method, we split a document based on its inherent structure. This approach considers the flow and structure of content but may not be as effective documents lacking clear structure.



Chunking Strategies

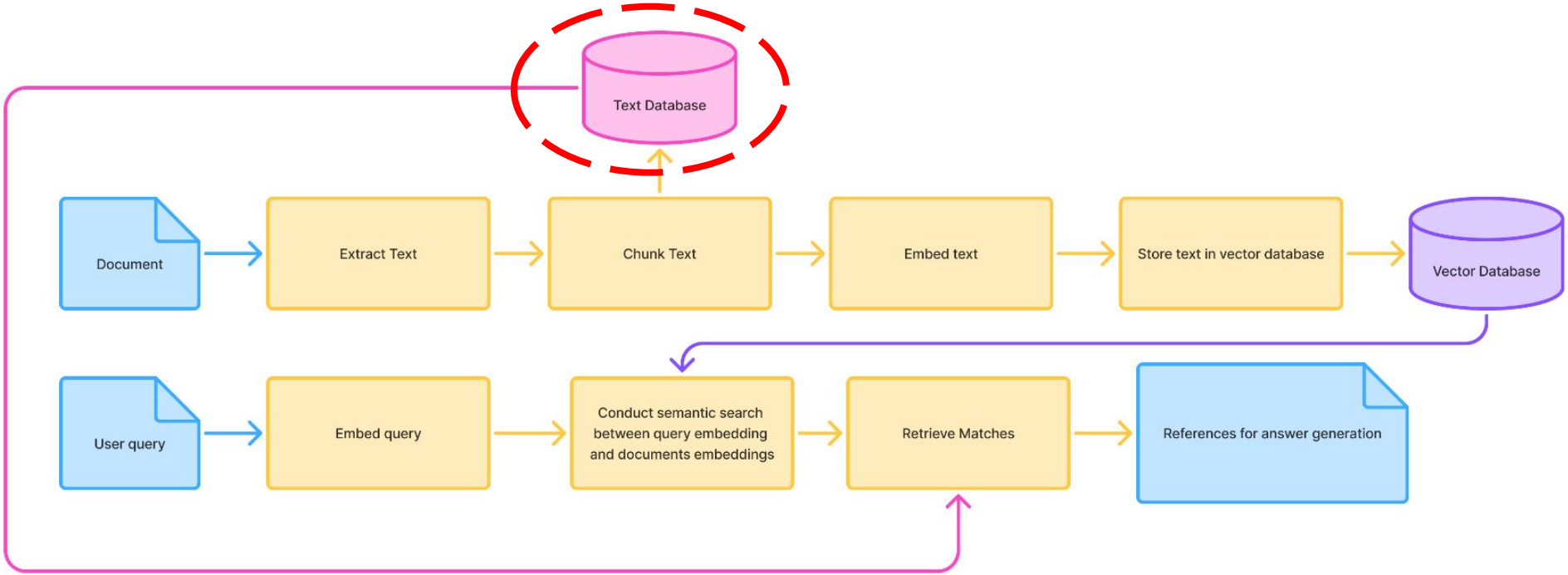
5. LLM-based Chunking: This chunking strategy explore the possibility to use LLM to determine how much and what text should be included in a chunk based on the context.



Hands on: 4_Chunking_Strategies.ipynb

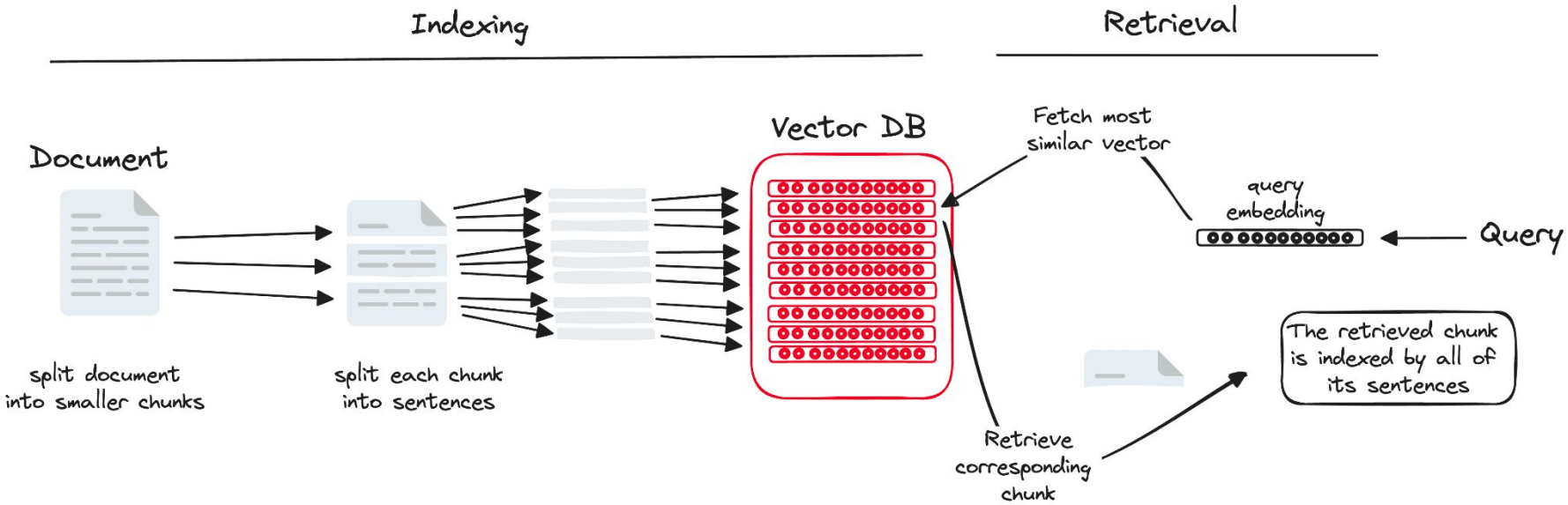
Before we continue to process the chunks we need to index and save them.

RAG- The basic pipeline



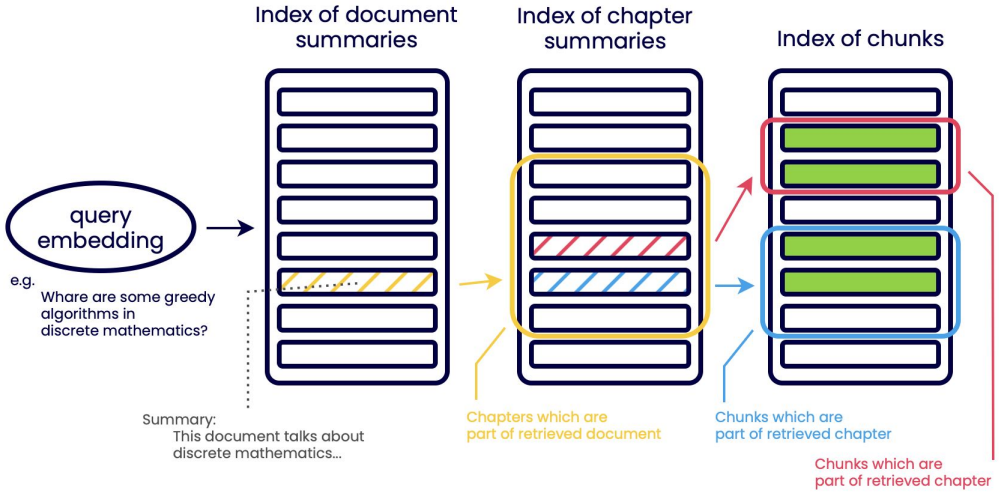
Why do we save chunks?

- When we **match** against the **embedding** of a chunk, we need to use it's **reference index** to fetch it and append it to the prompt.



Layers of Indexing

- **Indexing** can be more complicated than fetching the chunk, which would let us know which document the answer is in.
- In some **RAG applications**, we have **multiple indices**.



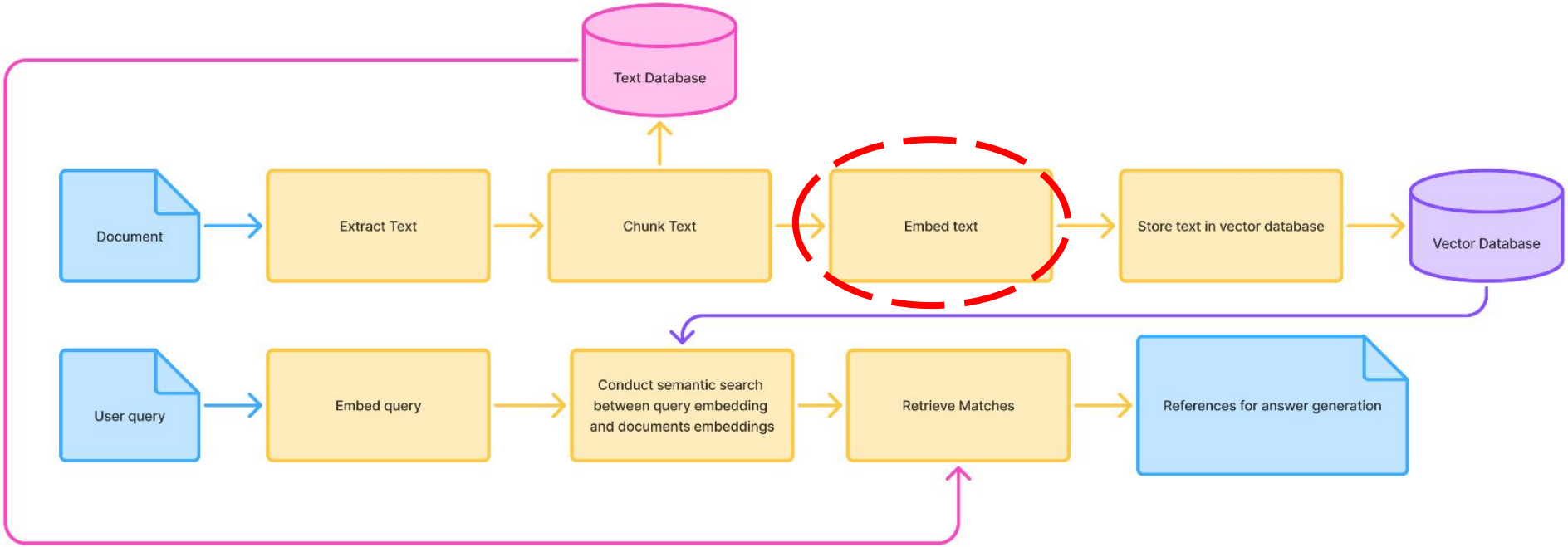
Saving Chunks

- You can save the chunks in any type of database you prefer.
- It can **relational** or **unstructured**.

```
chunk = {
  'chunk_id' = 1
  'original_document' = 'Attention is all you need'
  'chunk' = 'The encoder is composed of a stack of N = 6 identical layers. Each layer has two sub layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network.'
}
```

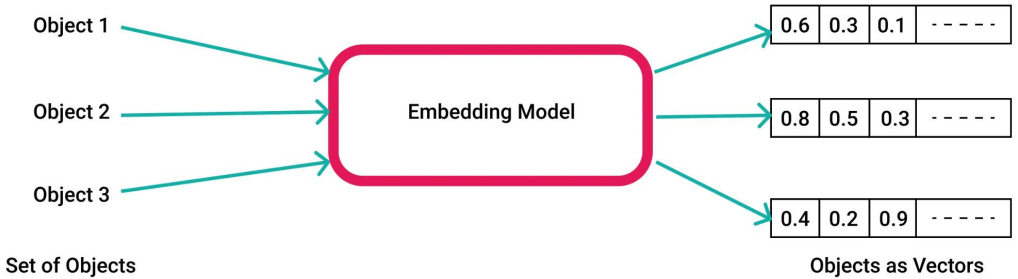
Now that we have created and saved our chunks, we need to embed them.

RAG- The basic pipeline



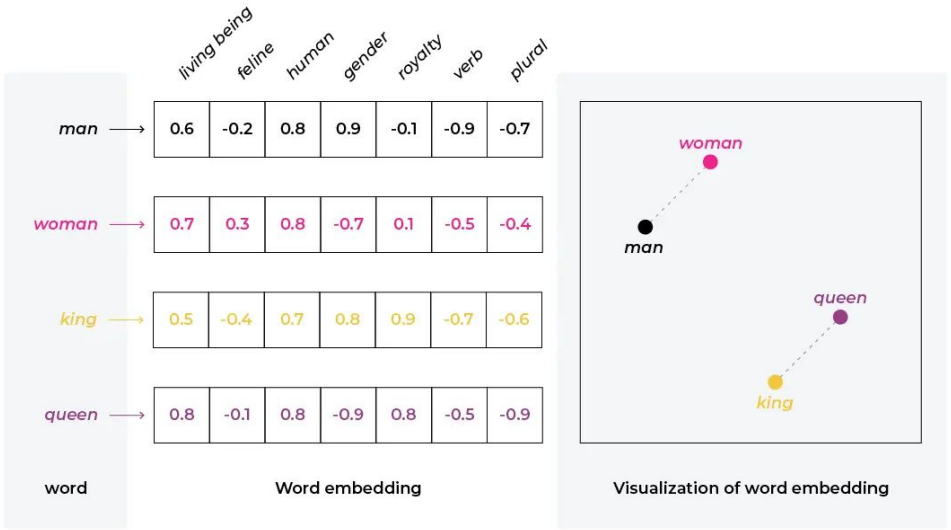
Recap: Embeddings

- Embeddings are mathematical **vector representations of real-world data**, such as text, images, and audio, used by machine learning models to **understand semantic meaning and relationships**.
- These low-dimensional vectors capture the properties of the original data in a compact, numerical form, allowing AI systems to **perform tasks like similarity search.**



Recap: Embeddings

- These **numerical representations** capture the underlying **semantic meaning** and **inherent relationships** of the data.



Embedding Models

Model type	Embeddings	Example model	Compute	Granularity/input
Sparse	0.3 [ ] 0.8	SPLADE	Low	Sentence, paragraph
Dense	0.3 0.1 0.8	Sentence transformers	Medium	Sentence, paragraph
Multivector	0.4 0.3 0.2 x tokens	COLBERT	High	Sentence, paragraph
Long context dense	0.3 0.1 0.8	text-embedding-3-small	Medium	Paragraphs, chapters
Variable dimension	0.3 0.1 0.8 Dimensions can change	text-embedding-3-small	Medium	Sentence, paragraph
Code (dense)	0.3 0.1 0.8	text-embedding-3-small	Medium	Functions, classes

Embedding Models

Smaller Embedding Models (e.g., OpenAI text-embedding-3-small)

Pros:

1. **Faster speed** (lower latency): They generate embeddings more quickly, which is crucial for real-time applications like chatbots and recommendation systems.
2. **Reduced storage and cost**: Smaller model sizes mean less storage is required for embeddings, leading to lower costs for vector databases and overall infrastructure.
3. **Efficiency:** They consume less computational power and memory, making them more accessible for use on less powerful hardware or for on-device applications.

Cons:

1. **Potentially lower accuracy:** They might not capture as much semantic

Embedding Models

Larger Embedding Models (e.g., OpenAI text-embedding-3-large)

Pros:

1. **Higher accuracy and expressiveness:** They can better understand complex relationships and patterns in text, producing more context-aware and detailed embeddings.
2. **Better performance on benchmarks:** They often achieve higher scores on tasks like semantic search and question answering.

Cons:

1. **Slower speed (higher latency):** Generating embeddings takes more time due to the model's complexity and higher dimensionality.
2. **Increased resource consumption:** They require more memory and storage, leading to higher costs for deployment and use.

Embedding Models

Top Rated Embedding Models For RAG

Open Source

BGE Models:
Developed by the Beijing Academy of Artificial Intelligence (BAAI)

E5 Family:
Models such as intfloat/e5-large-v2 and its multilingual counterpart multilingual-e5-large are strong contenders

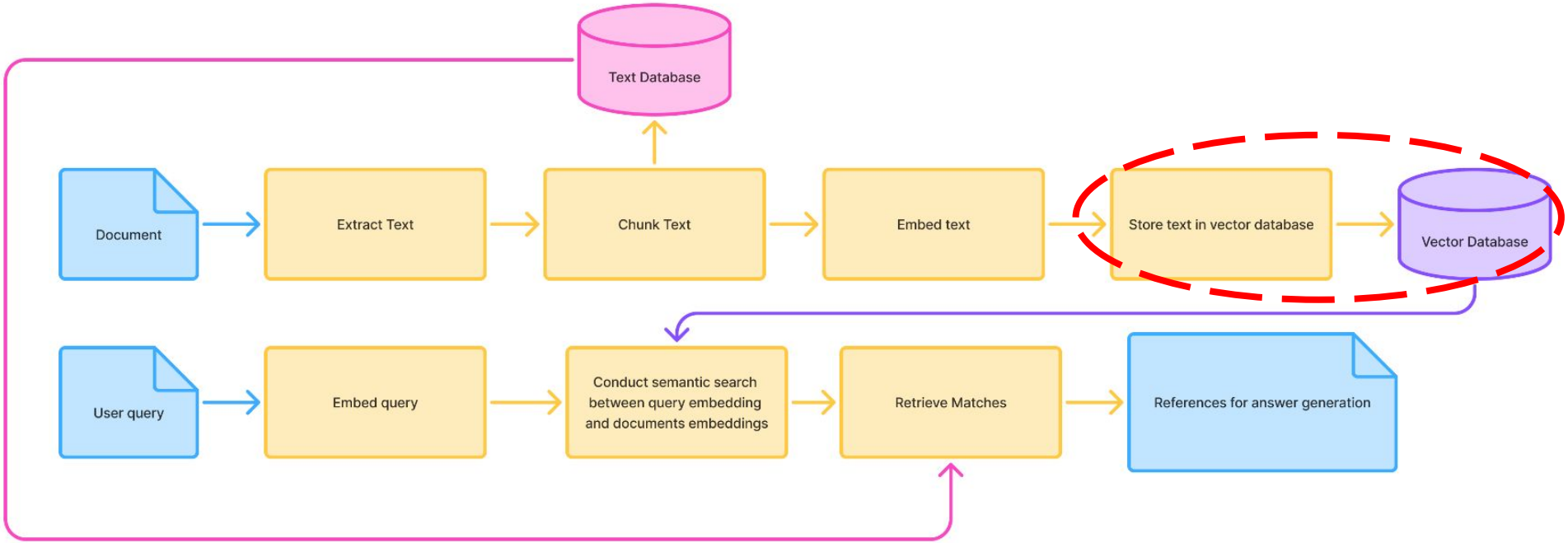
Closed Source

- OpenAI Embeddings: Models like text-embedding-3-small, text-embedding-3-large, and older models like ada-002 are popular for RAG. They are known for strong performance but come with usage costs.

Hands On: 5_Dense_Embedding_Models_for_RAG.ipynb

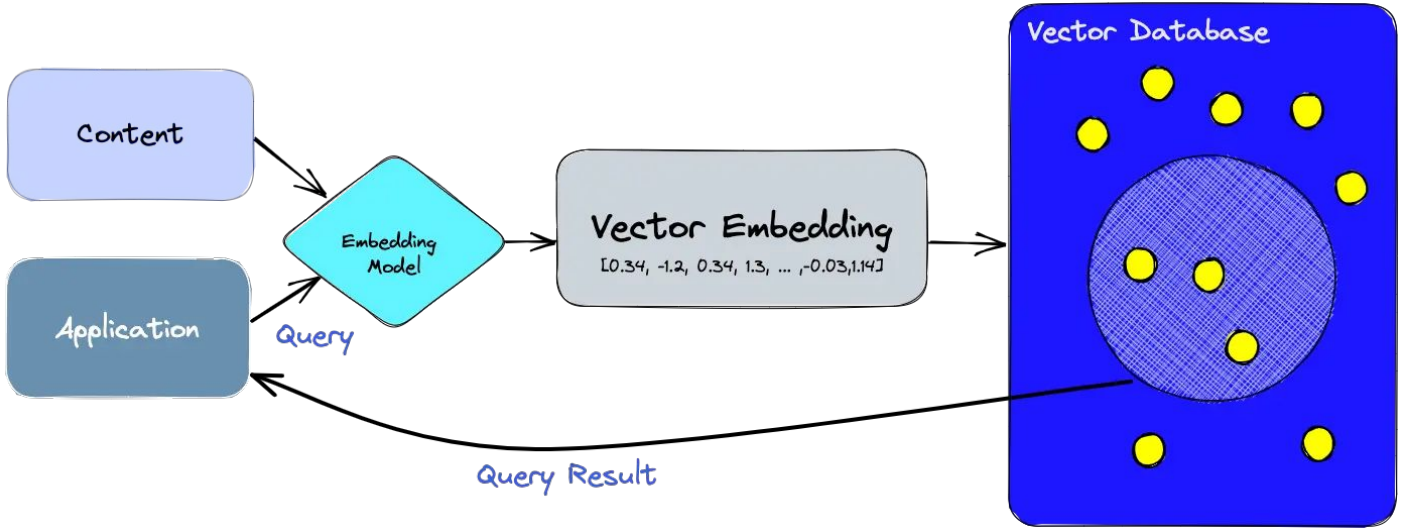
Now that we have our chunk's embeddings. We need to store them somewhere to search against when the query arrives.

RAG- The basic pipeline



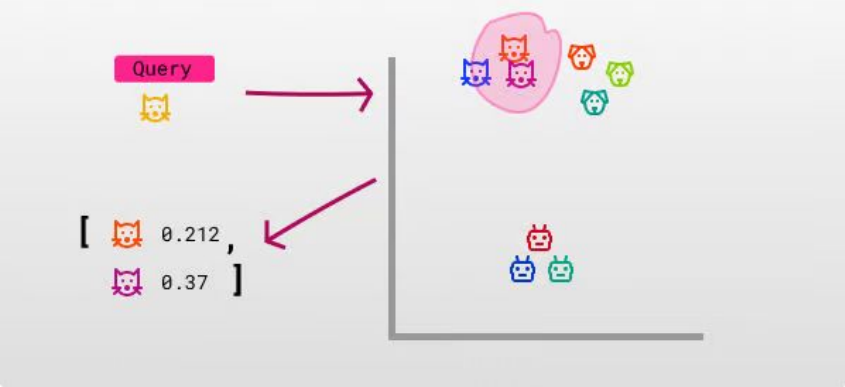
What Is a Vector Database?

- A specialised database that **stores data as high-dimensional vectors** (numerical arrays) that enables **fast similarity search** and retrieval of complex data like text, images, and audio

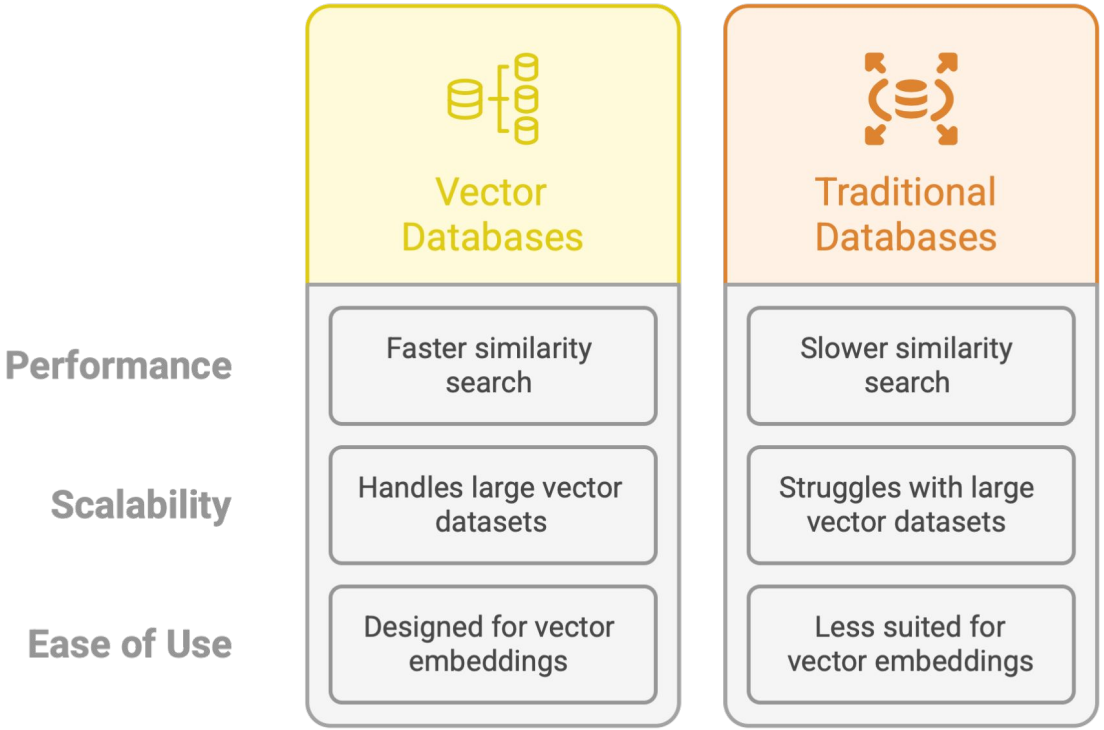


Why Traditional Databases Fall Short

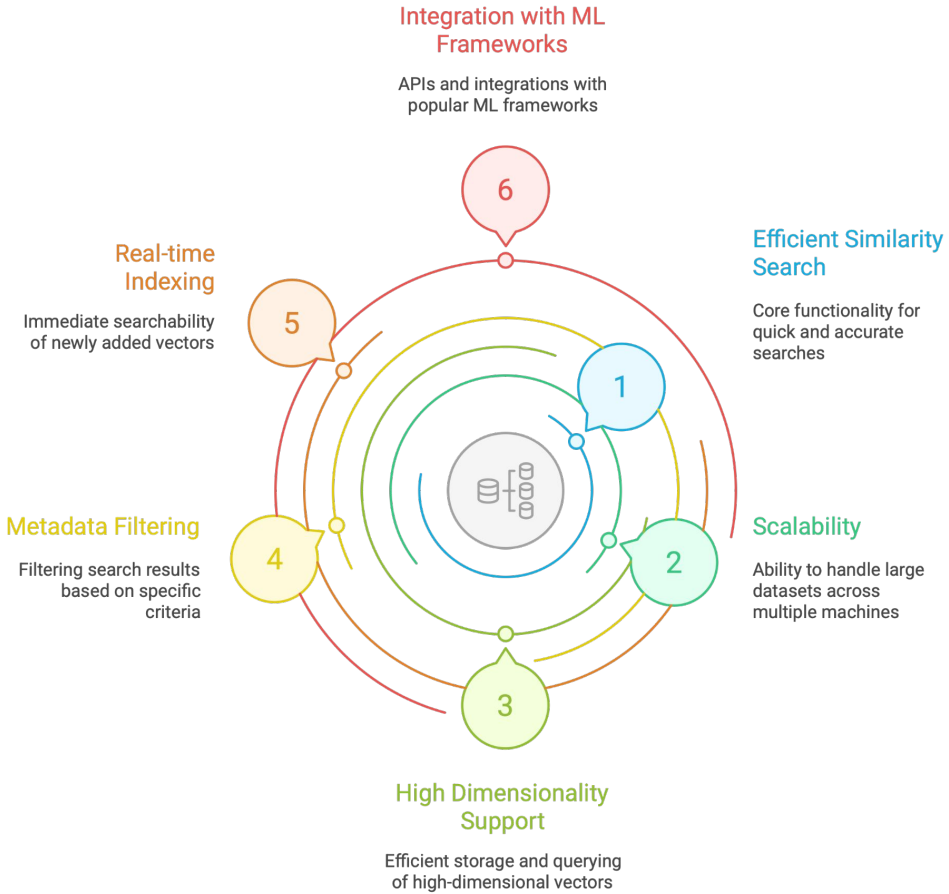
- Conventional databases store **scalar values** (numbers, strings) and rely on exact matches
- Unable to efficiently handle **multi-dimensional, semantic-rich data** from AI models
- Vector databases are purpose-built to **manage** and **query vector embeddings** at scale



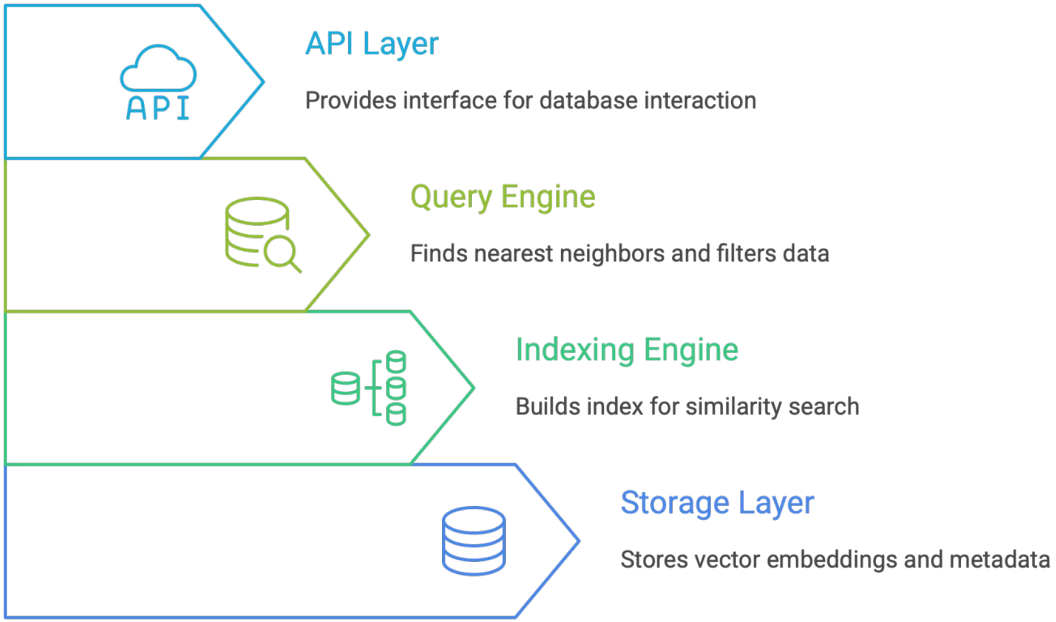
Why Traditional Databases Fall Short



Core Technologies Behind Vector Search

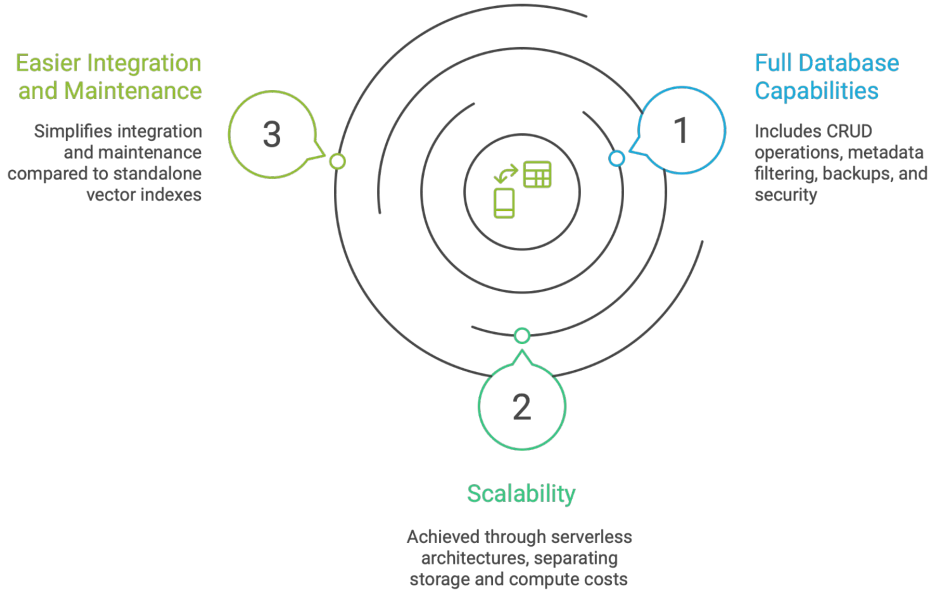


Vector Databases Under The Hood



Advantages Over Standalone Vector Indexes

- In the RAG hands on, we saw that we can use a vector index instead of a full vector store to store out embeddings. Why would we need a **vector store** then?

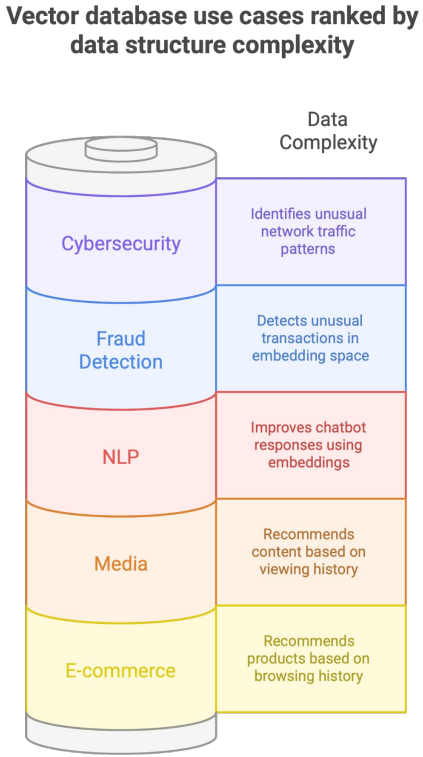


Real-World Use Cases

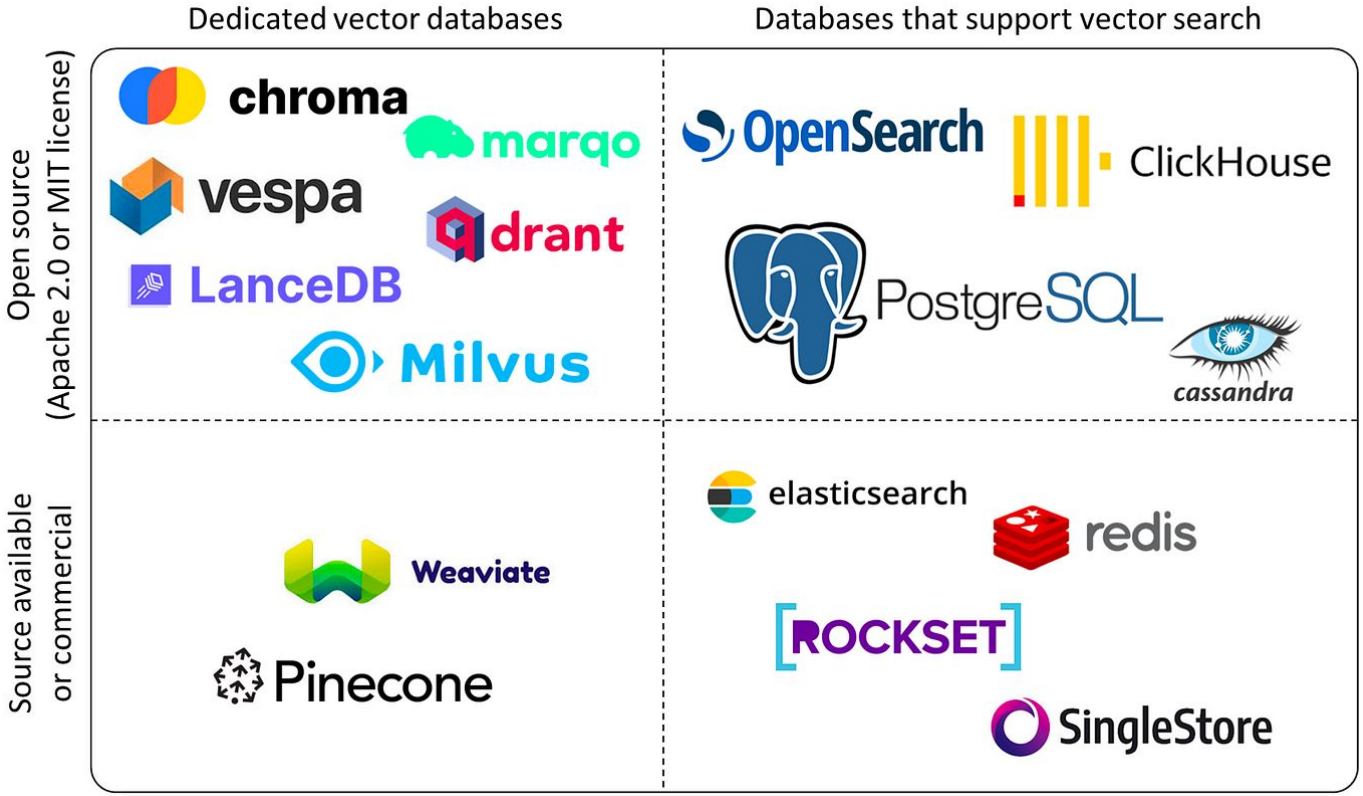
We are seeing the next generation of vector databases introduce more sophisticated architectures to handle the efficient cost and scaling of intelligence. This ability is handled by serverless vector databases, that can separate the cost of storage and compute to enable low-cost knowledge support for AI.

- **Semantic search**: finding documents or images by meaning, not keywords
- **Recommendation systems:** suggesting products, songs, or movies based on similarity
- Enhancing large language models with **long-term memory and contextual retrieval**

Industry Impact and Growth



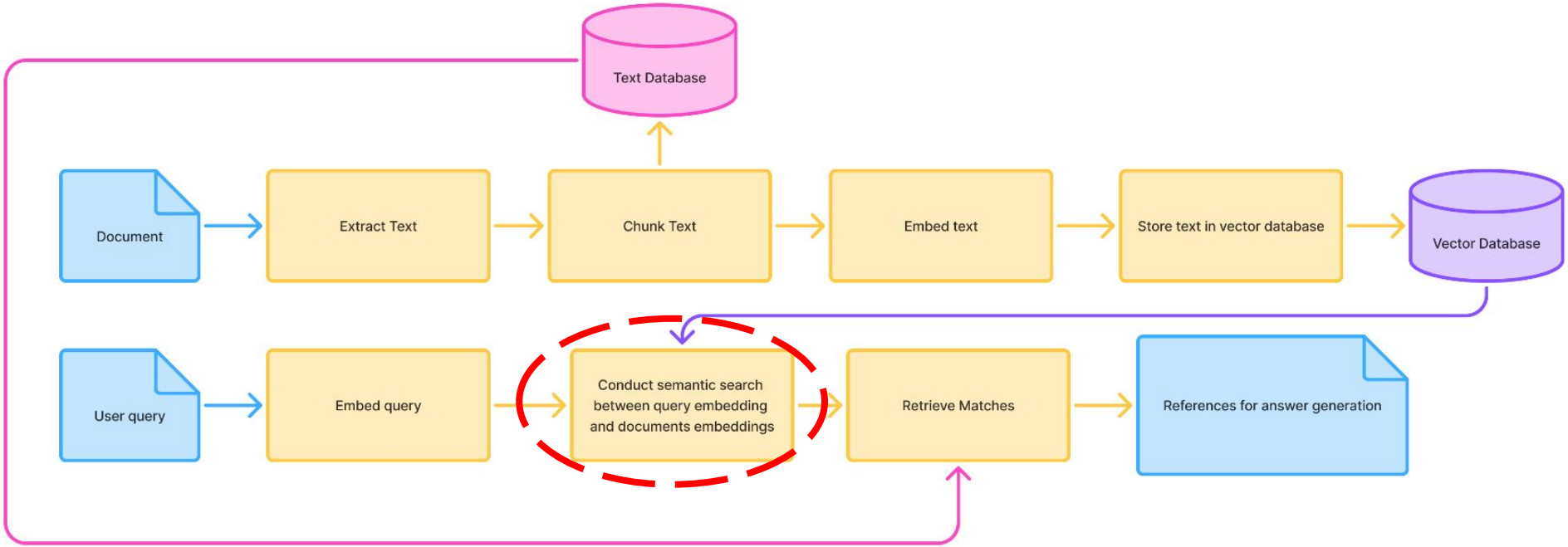
Popular Vector Databases



Hands on: 6_Vector_Stores_for_Similarity_Search.ipynb

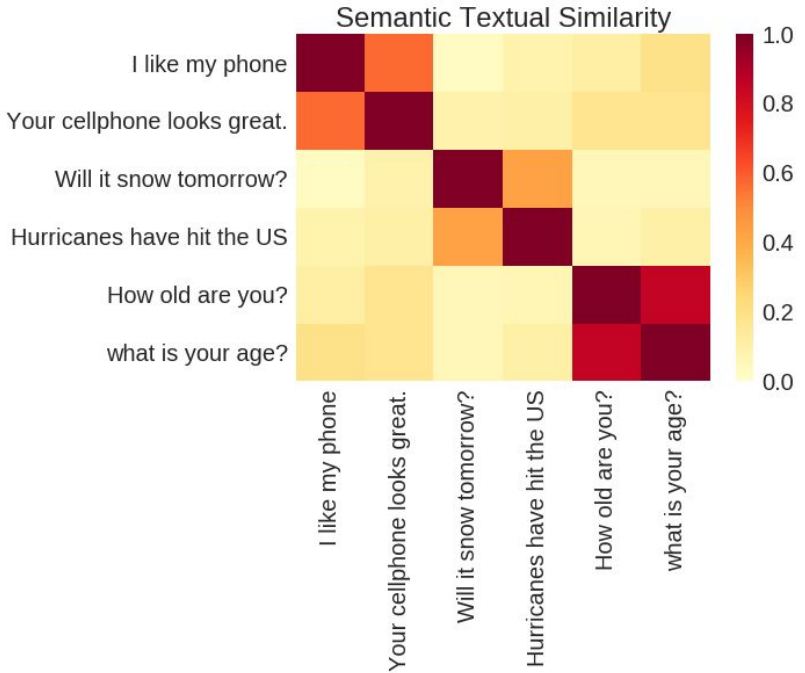
Semantic similarity

RAG- The basic pipeline



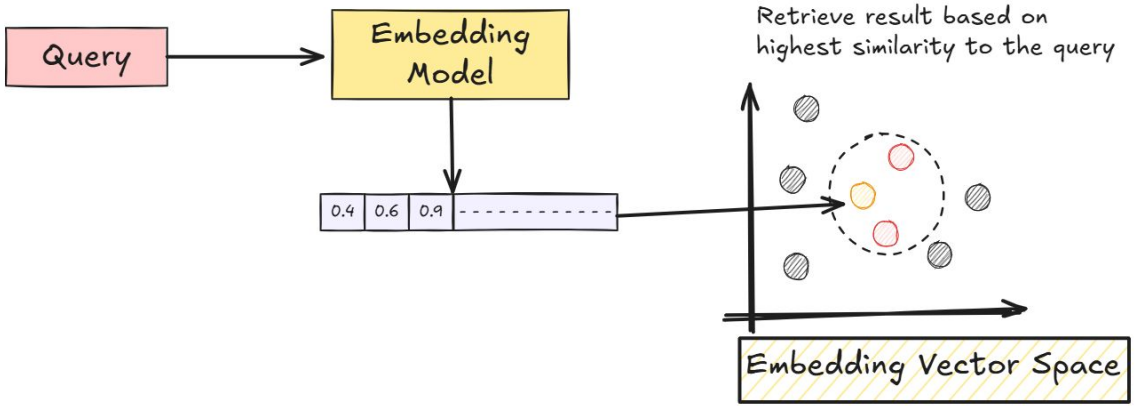
The Role of Semantic Similarity in RAG

1. Goes beyond keyword matching to **capture meaning** and **context** between query and documents
2. Enables retrieval of **conceptually relevant** text even if exact words differ
3. Core to finding the **best context chunks** to feed into the LLM for accurate answers



How do you compare embeddings?

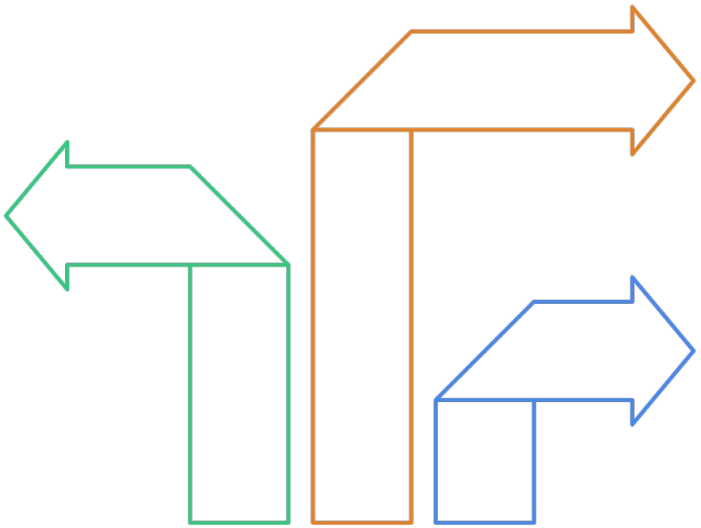
1. The concept of Answer Semantic Similarity pertains to the assessment of the semantic **resemblance between the generated answer and the ground truth**, with values falling within the **range of 0 to 1.**
2. A higher score signifies a **better alignment** between the generated answer and the ground truth.



How do you compare embeddings?

Euclidean Distance

Best for straight-line distance in multi-dimensional space.



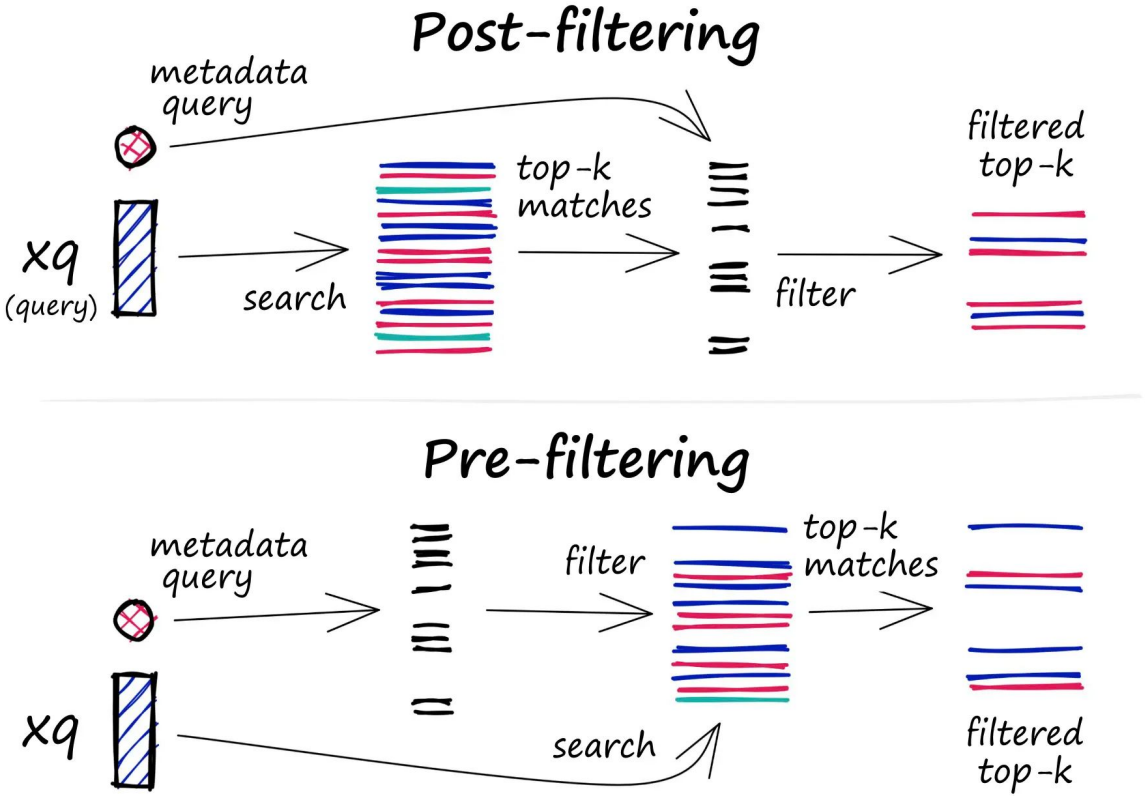
Dot Product

Useful for measuring vector alignment, a component of cosine similarity.

Cosine Similarity

Ideal for text analysis as it focuses on angle, not magnitude.

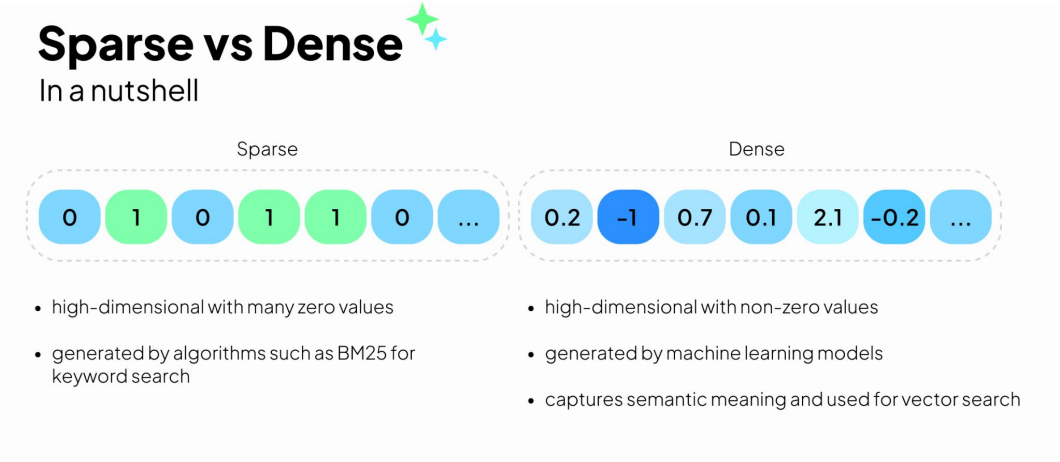
Filtering Matches and Results



Dense Embeddings are the most popular type of embeddings used, but it is not the only one!

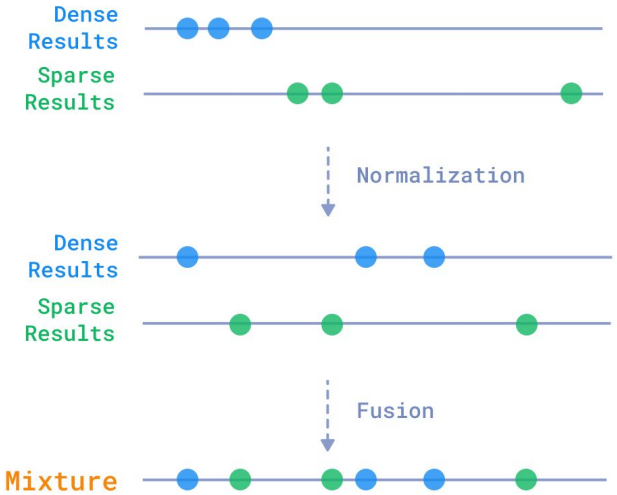
Sparse Embeddings

- These embeddings are generally high-dimensional, with **most dimensions inactive or zero**.
- Sparse embeddings were among the earliest forms of word representation in NLP, prominently seen in techniques like one-hot encoding and bag-of-words models.

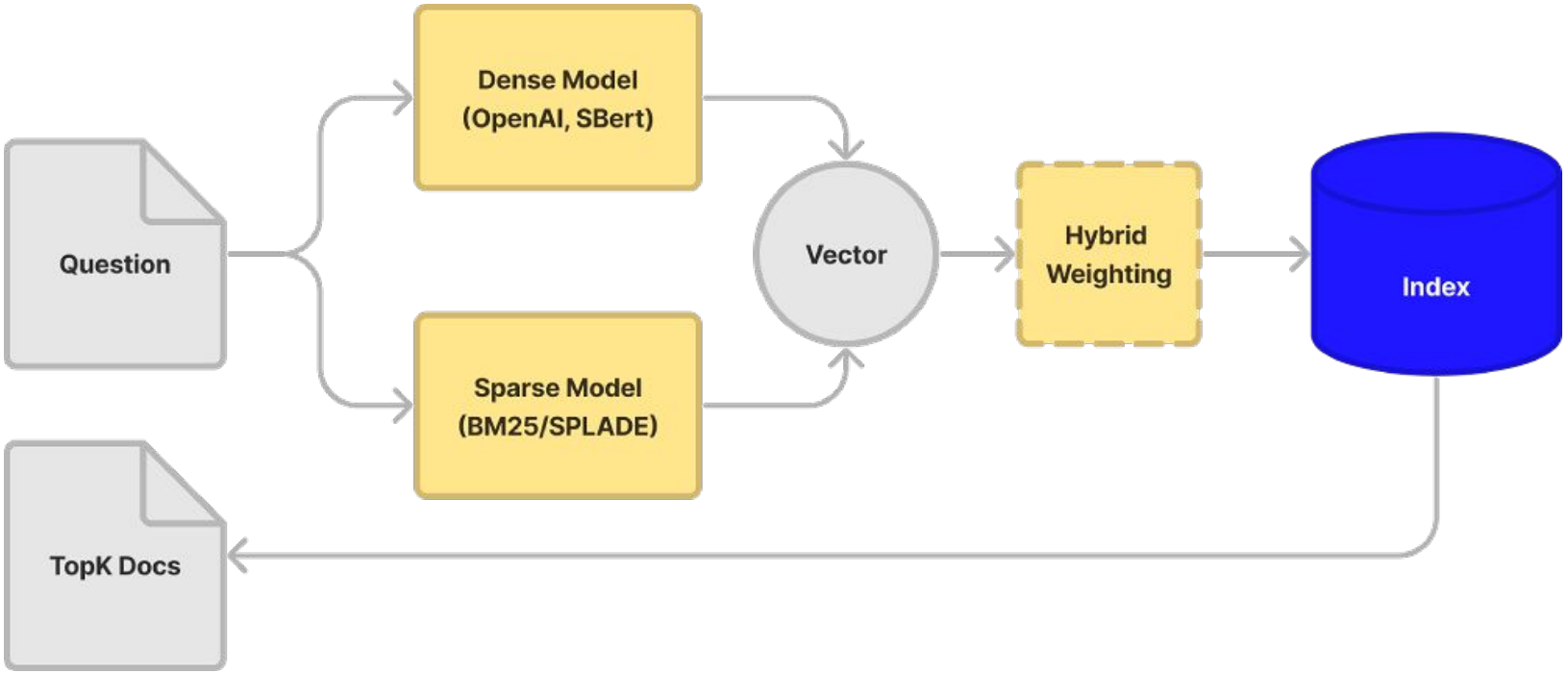


Sparse Embeddings

- Sparse embeddings are very useful to ensure **keyword matching** between the query and the corpus.
- A hybrid approach can be used to combine **context matching** from dense embeddings and keyword matching from sparse embeddings

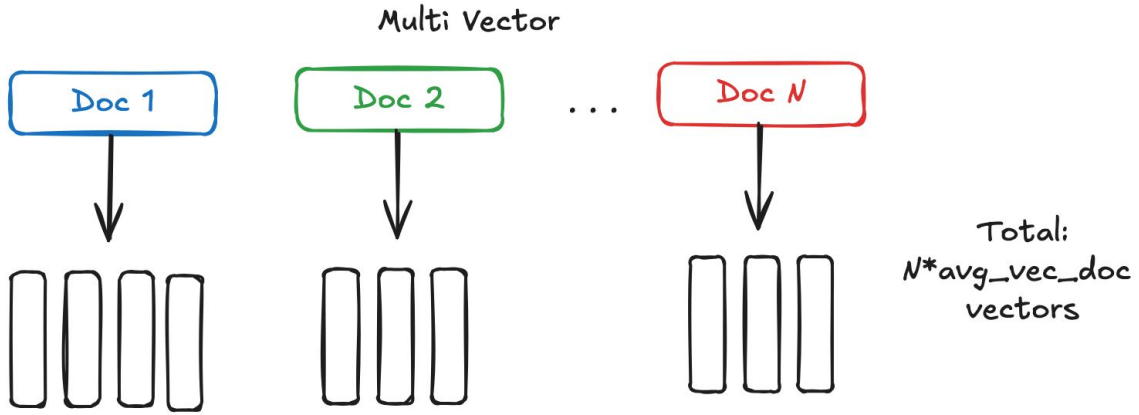


Sparse Embeddings



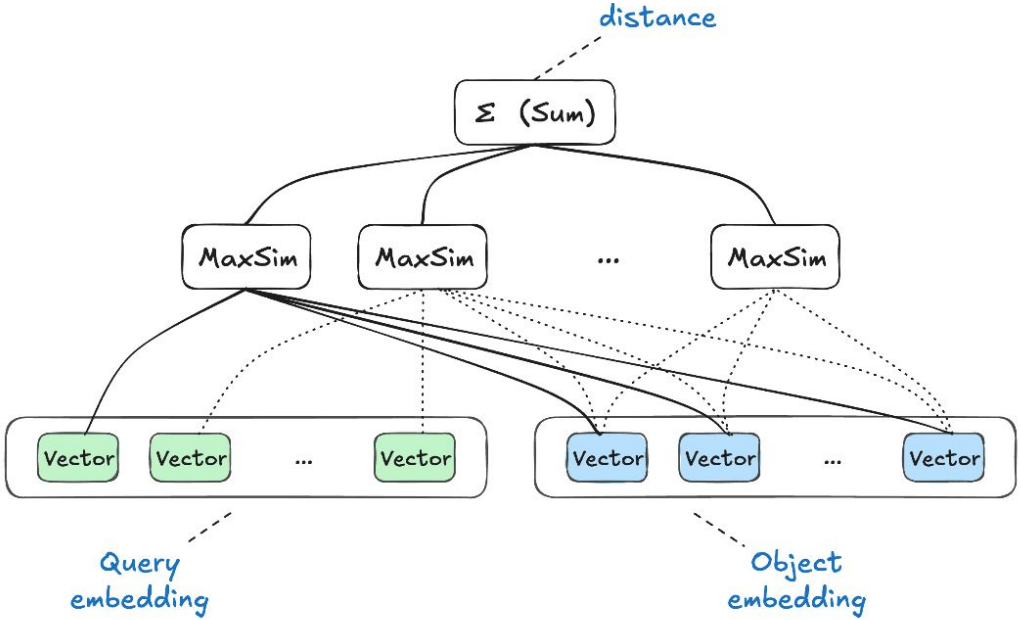
Multi-vector Embeddings

- Multi-vector embeddings **represent a single data object** (like a document or image) with a **collection of multiple smaller vectors,** where each vector **captures specific aspects or segments of the object**, allowing for more detailed semantic information capture compared to single-vector approaches.



Multi-vector Embeddings

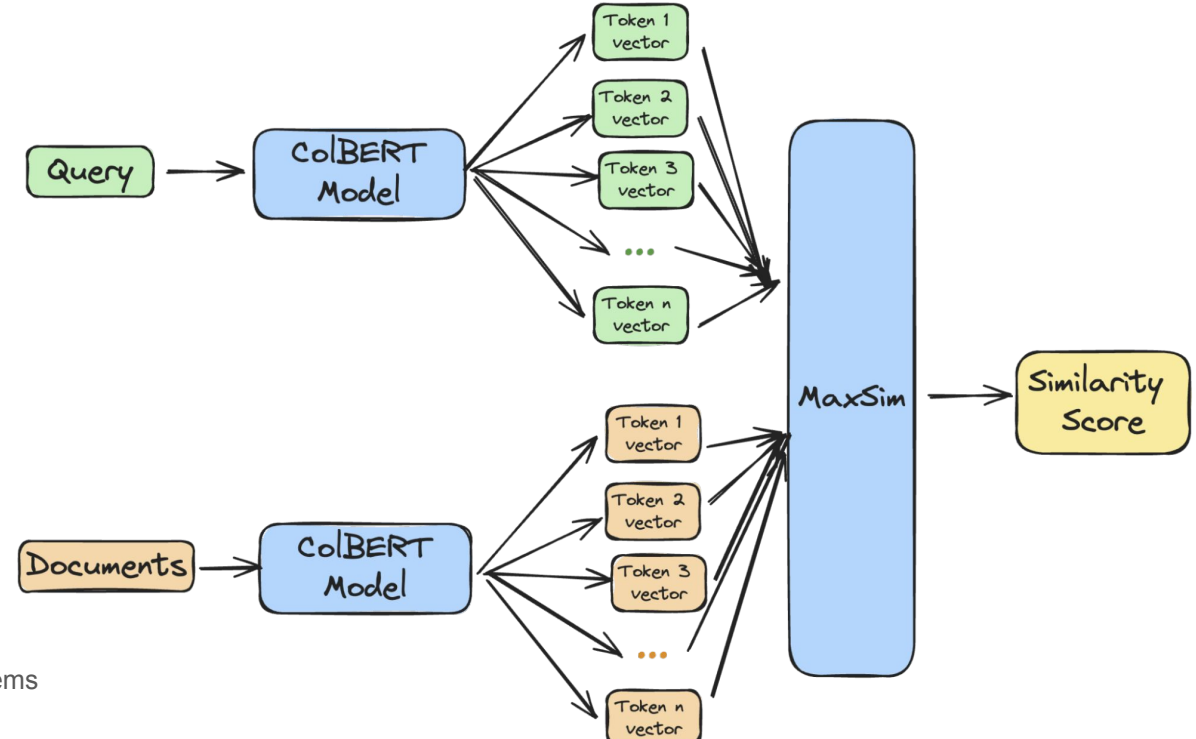
- While this offers **superior accuracy**, it significantly **increases memory usage** and computational costs due to the greater number of vectors.



with multi-vector embeddings

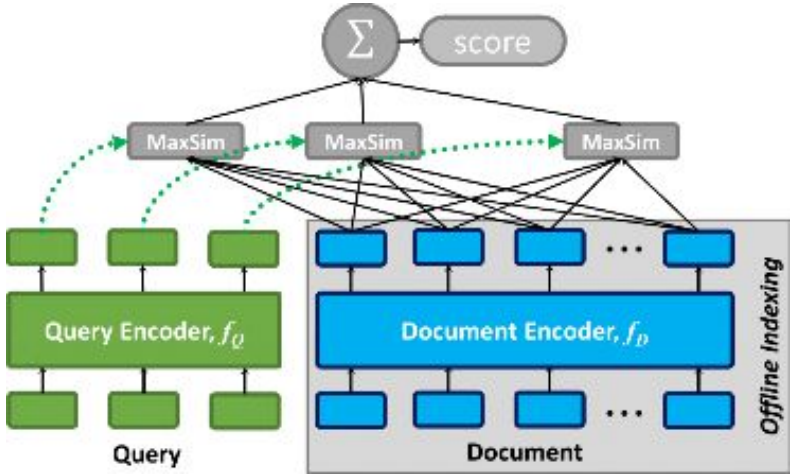
Multi-vector Embeddings

- The most famous **multi-vector embedding** model is **ColBert**.



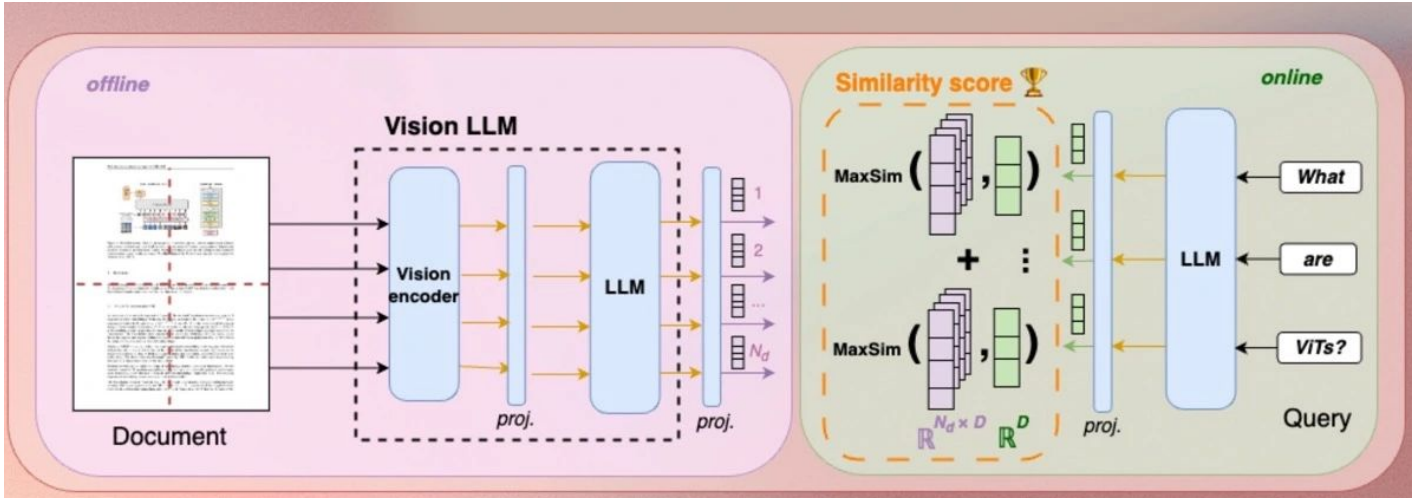
Multi-vector Embeddings

- ColBERT relies on fine-grained contextual **late interaction**: it encodes each passage into a matrix of **token-level embeddings** (shown above in blue).
- Then at search time, it embeds every query into another matrix (shown in green) and efficiently **finds passages that contextually match the query using scalable vector-similarity (MaxSim)** operators.

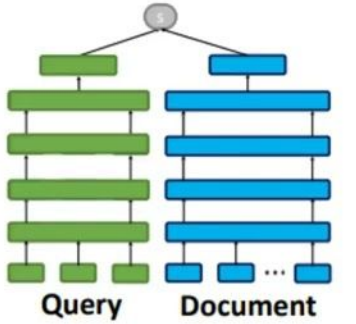


Multi-vector Embeddings

- Multi-vector embeddings have been used for efficiently performing semantic similarity between **text queries** and **embedded images** using the ColPali Embedding model



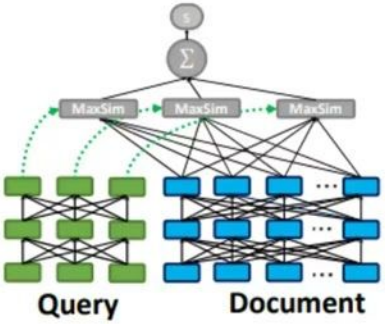
Multi-vector Embeddings



(a) Representation-based Similarity
(e.g., DSSM, SNRM)

Bi-encoders

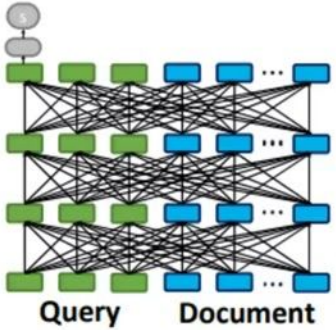
- Fast and simple
- Indexing can be offline



(d) Late Interaction
(i.e., the proposed ColBERT)

Late-interaction

- Richer signal (multi-vector)
- Indexing can be offline



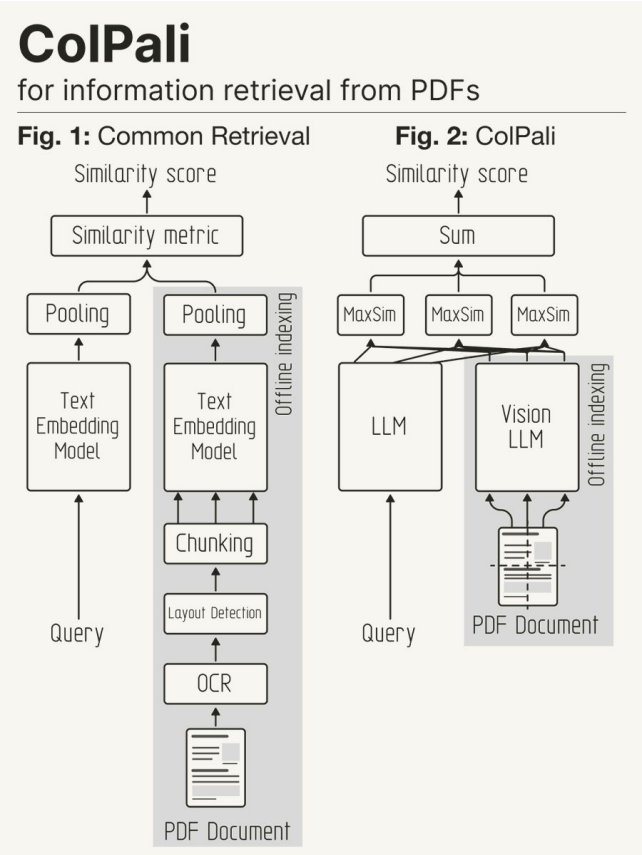
(c) All-to-all Interaction
(e.g., BERT)

Cross-encoders

- Very rich signal
- Indexing is online and expensive

Khattab, O., & Zaharia, M. (2020). ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT.

Multi-vector Embeddings



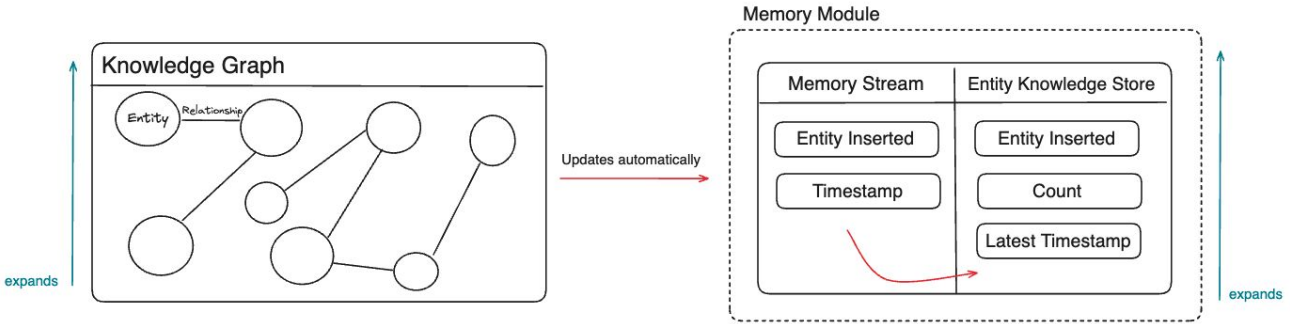
Hands on: 7_Semantic_Similarity.ipynb

Different Ways of Modeling the Agent's Memory

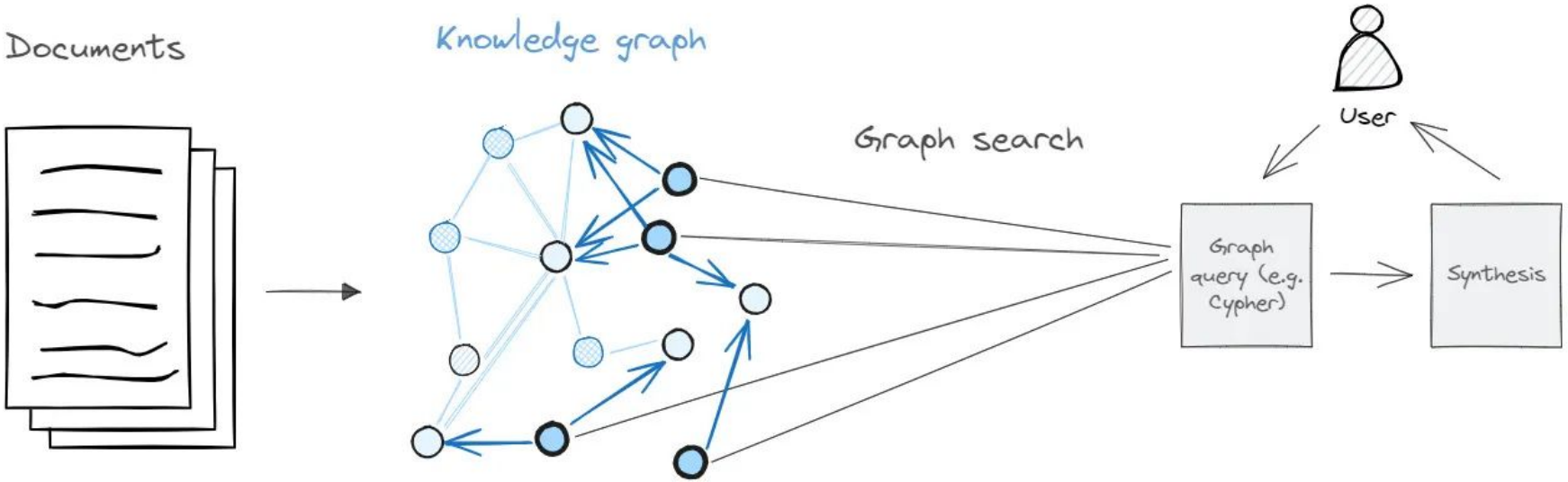
Now that we have discussed how to ingest any document into an agent's memory. Let's different ways of modeling the agent's memory.

Knowledge Graphs as Agent Memory

- Vector similarity, while powerful, captures only semantic similarity.
- Two pieces of text might be semantically similar but temporally unrelated, or vice versa.
- What we need is a system that can understand not just semantic meaning, but also **temporal context, entity relationships,** and **the evolution of knowledge over time.**



Knowledge Graphs as Agent Memory



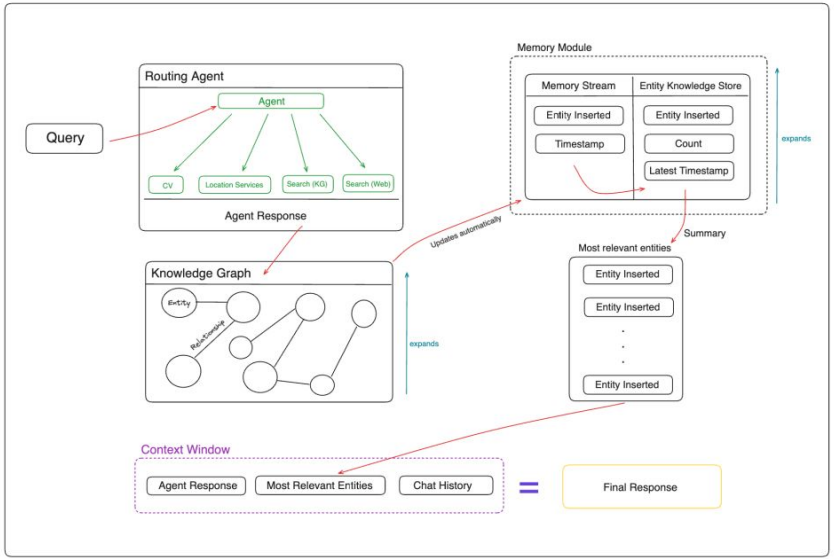
Knowledge Graphs as Agent Memory

The challenge isn't just about finding relevant information - it's about understanding how information connects, evolves, and relates across time and context. Traditional approaches struggle with:

1. **Temporal reasoning**: Understanding when events occurred and how they relate chronologically
2. **Entity disambiguation**: Distinguishing between different people, places, or things with similar names
3. **Relationship modeling**: Connecting related information that may not be semantically similar
4. **Context evolution:** Tracking how entities and relationships change over time

Knowledge Graphs as Agent Memory

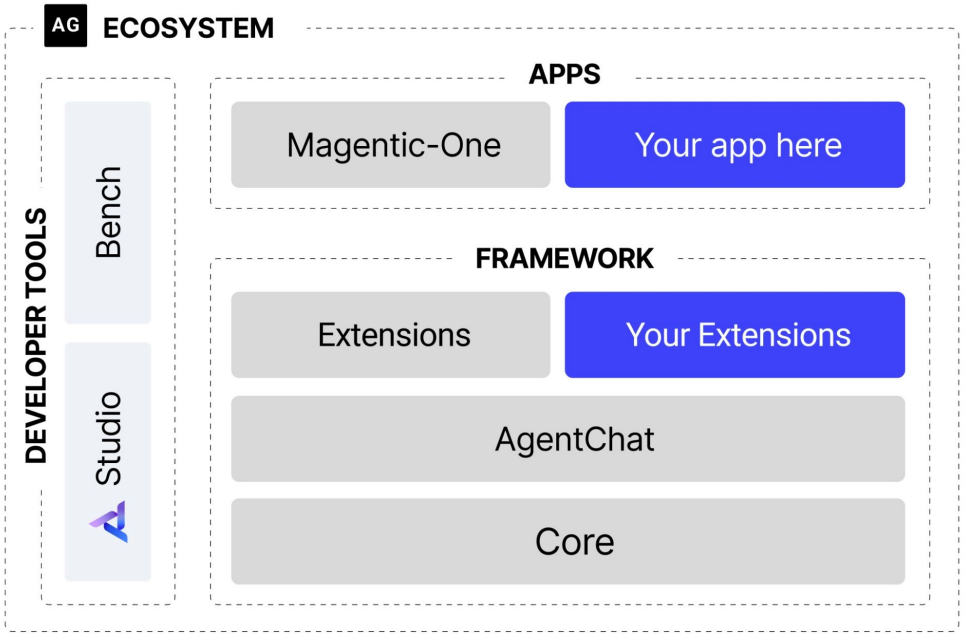
- A knowledge graph is a **structured representation** of real-world entities and their relationships.
- Unlike traditional databases that store data in tables or documents, knowledge graphs model information as **networks of interconnected nodes** (entities) and edges (relationships).



Now that we've seen how RAG can help us retrieve relevant information from long-term memory. Let's check how open source frameworks have implemented memory.

AutoGen

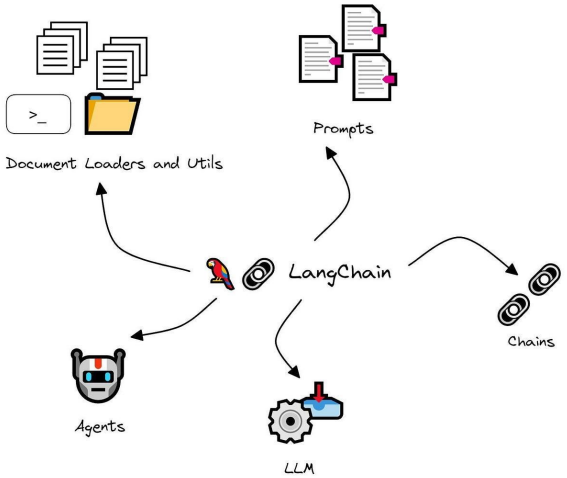
- AutoGen, created by Microsoft, is a **framework** for creating multi-agent AI applications that can act autonomously or work alongside humans.



Hands on: 8_AutoGen_Memory.ipynb

Langchain

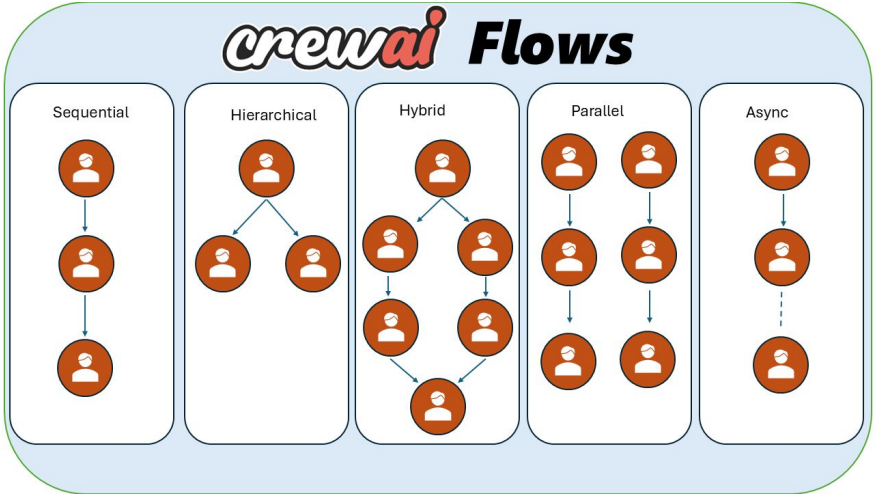
- **LangChain** provides a modular approach to agent memory, with a suite of **Memory classes** that can be mixed and matched.
- For short-term conversational memory, LangChain offers s**imple in-memory buffers** as well as summarization-based memory



Hands on:
9_LangChain_Agent_Memory.ipynb

CrewAI

CrewAI is a newer framework geared towards multi-agent orchestration, where each agent in a “crew” can have specific roles (e.g., a Developer agent, a Tester agent, etc.). CrewAI makes memory a first-class component of agent design. It provides **three distinct memory approaches** out-of-the-box. Under the hood, in the default Basic system uses short term memory, long term memory, and entity memory.

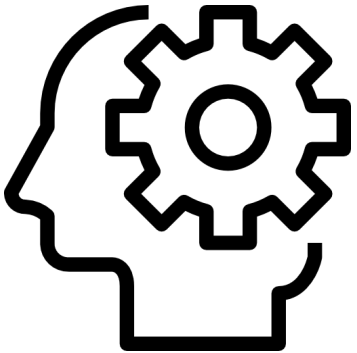


Hands on:
10_CrewAI_MultiAgent_Memory.ipynb

Agent Memory for Different Tasks

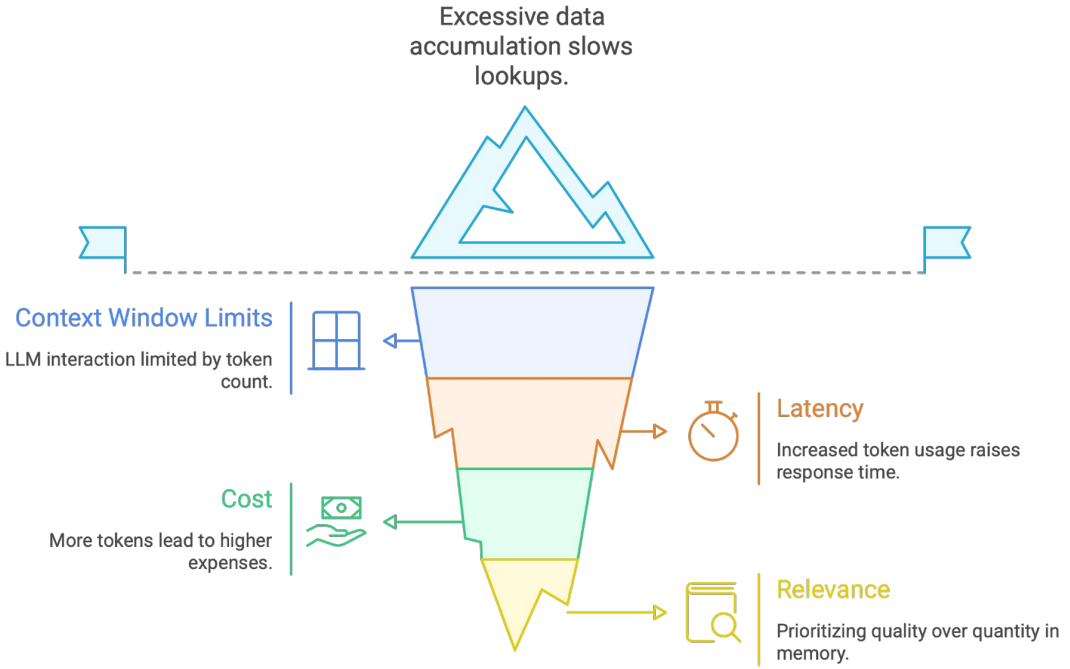
The trend in these frameworks has been toward **specializing memory to the task at hand**:

- If the agent is solving coding problems, memory might be file-based (saving code snippets or notes in files it can read later).
- If the agent is web-browsing, it might save links or page texts in a local cache.
- Multi-session long-term memory (like remembering a user from day to day) was not the initial focus of AutoGPT or BabyAGI, which were often run on single tasks.

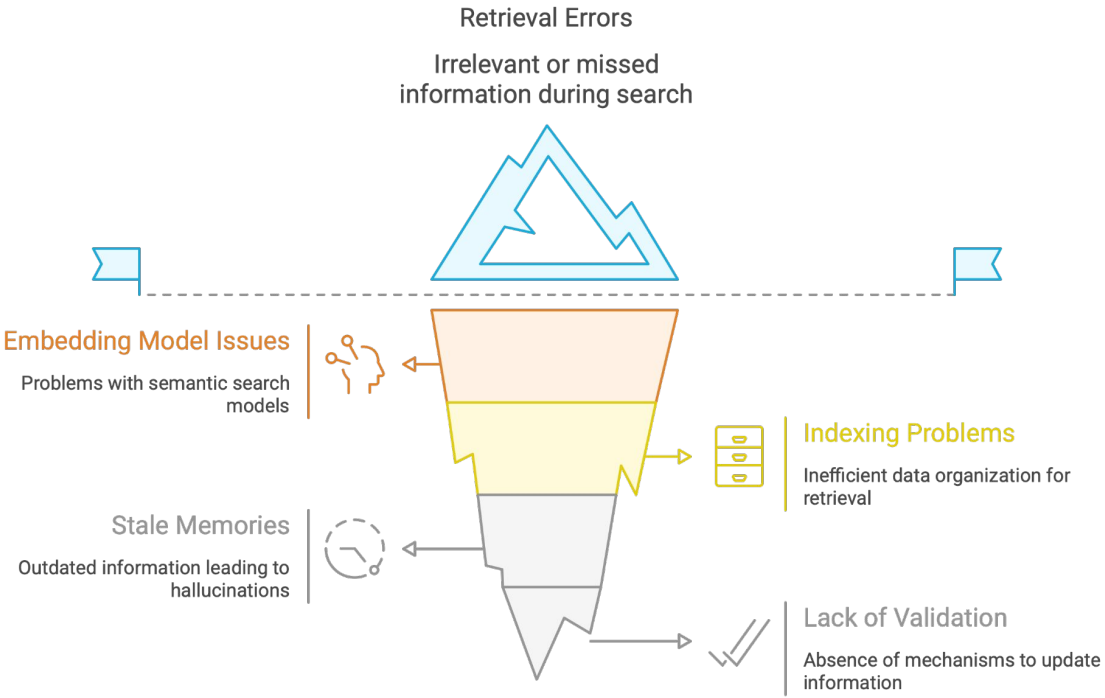


Challenges of adding memory to consider as your system scales

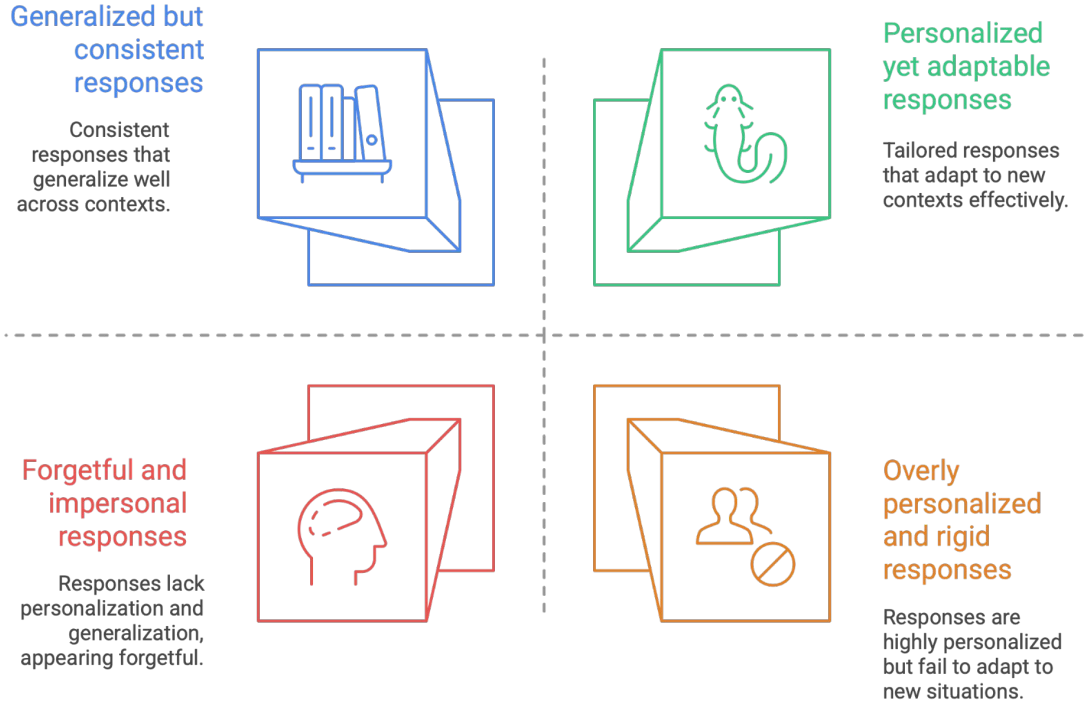
Memory Bloat



Retrieval Errors

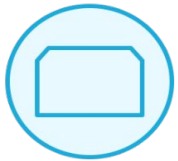


Balancing AI Memory Strategy



Memory Best Practices

Memory Management Process



Define Memory Boundaries

Establish clear scopes for memory

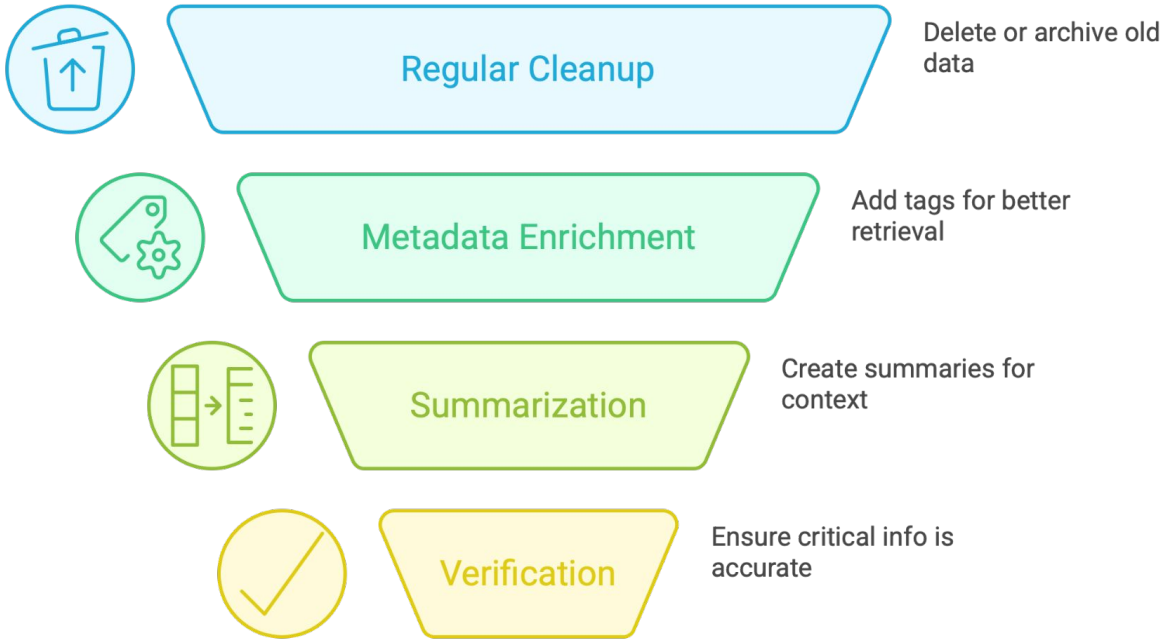
Store Relevant Information

Filter and store only necessary data

Implement Multi-Tier Memory

Combine short-term and long-term memory strategies

Memory Management Process



Enhancing Memory Interactions

Implicit Memory Access

Unstructured, untracked memory retrieval

Modularize Memory Logic

Implement memory interactions via function calls

Enable Memory Updates

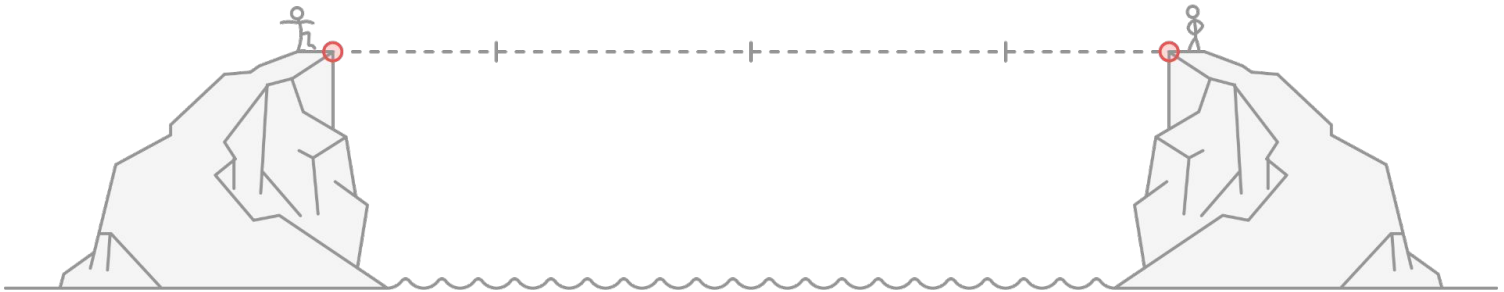
Correct old information with new data

Provide Memory Transparency

Allow users to inspect and edit memory

Explicit Memory Access

Structured, tracked, and editable memory



Enhancing LLM Memory Management

Unreliable LLM Memory

Inconsistent and error-prone retrieval

Monitor Memory Influence

Log memory usage for debugging

Test at Scale

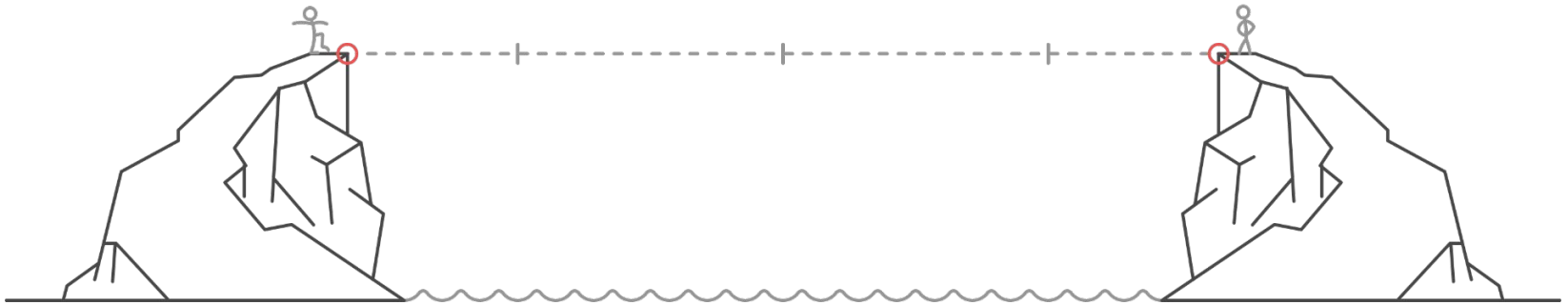
Simulate long interactions with synthetic memory

Stay Updated on Research

Implement proven techniques, watch emerging methods

Reliable LLM Memory

Consistent and intelligent information retrieval



In conclusion, **memory is the key to moving AI agents from merely reactive to truly proactive and adaptive systems**. By giving agents the ability to recall and learn from the past, we enable continuity, personalization, and the accumulation of knowledge over time, all essential for advanced autonomous behavior.

Yet, with great memory comes great responsibility: it must be managed carefully to remain an asset rather than a liability.

Thank You