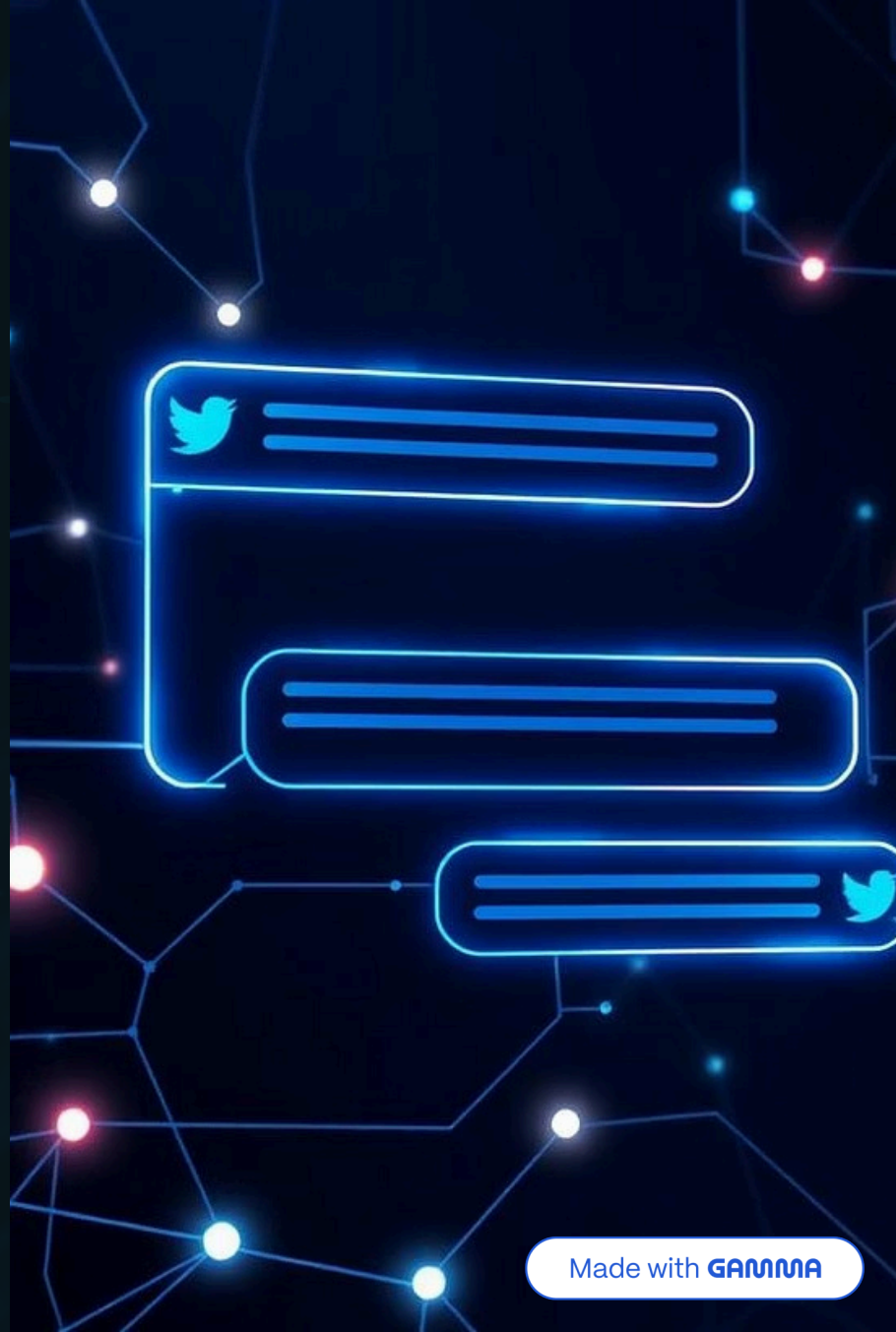
Building RAG Chatbots with LangChain

Explore the power of Retrieval Augmented Generation (RAG) to create intelligent and context-aware chatbots using the LangChain framework.



Agenda: A Deep Dive into RAG

1. **Understanding RAG Architecture**
   How RAG enhances LLMs with external knowledge.

2. **Introducing LangChain**
   Key components and their roles in RAG.

3. **Implementing Data Ingestion**
   Loading, splitting, and embedding your data.

4. **Building the Retrieval System**
   Vector stores and efficient similarity search.

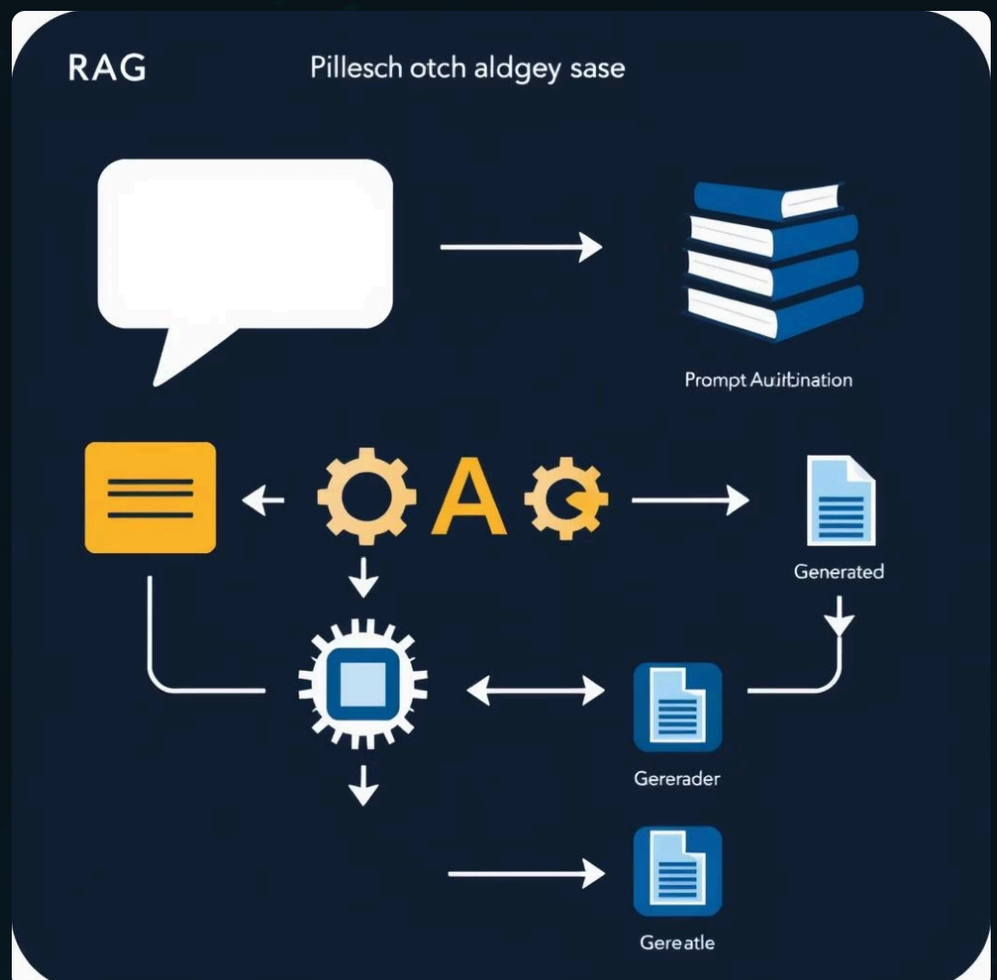
5. **Integrating with LLMs**
   Connecting retrieved context to language models.

6. **Advanced RAG Techniques**
   Exploring optimization and evaluation.

What is Retrieval Augmented Generation (RAG)?

RAG is an AI framework that retrieves factual information from an external knowledge base and uses it to ground a large language model's (LLM) response. This mitigates issues like hallucinations and provides up-to-date, relevant answers.

- Combats LLM hallucinations.
- Provides access to proprietary data.
- Enhances factual accuracy.
- Enables dynamic, real-time information.



LangChain: The RAG Orchestration Framework

LangChain simplifies the development of LLM-powered applications, particularly RAG systems, by providing modular components and chains to connect them.

Chains

Combine LLMs and other components for multi-step workflows.

Memory

Persist conversational state between turns.

Document Loaders

Ingest data from various sources (PDFs, websites).

Vector Stores

Store and query embeddings efficiently.

Data Ingestion: Preparing Your Knowledge Base

The first step in building a RAG chatbot is to load your data, split it into manageable chunks, and convert these chunks into numerical representations (embeddings).

1. Document Loading

Use LangChain's document loaders to ingest data from diverse sources, such as text files, PDFs, web pages, or databases. Each document is represented as a LangChain "Document" object.

Example:

```
PyPDFLoader("paper.pdf")
```

2. Text Splitting

Large documents are split into smaller, semantically meaningful chunks to optimize retrieval. LangChain offers various text splitters (e.g., RecursiveCharacterTextSplitter) to handle this.

Example:

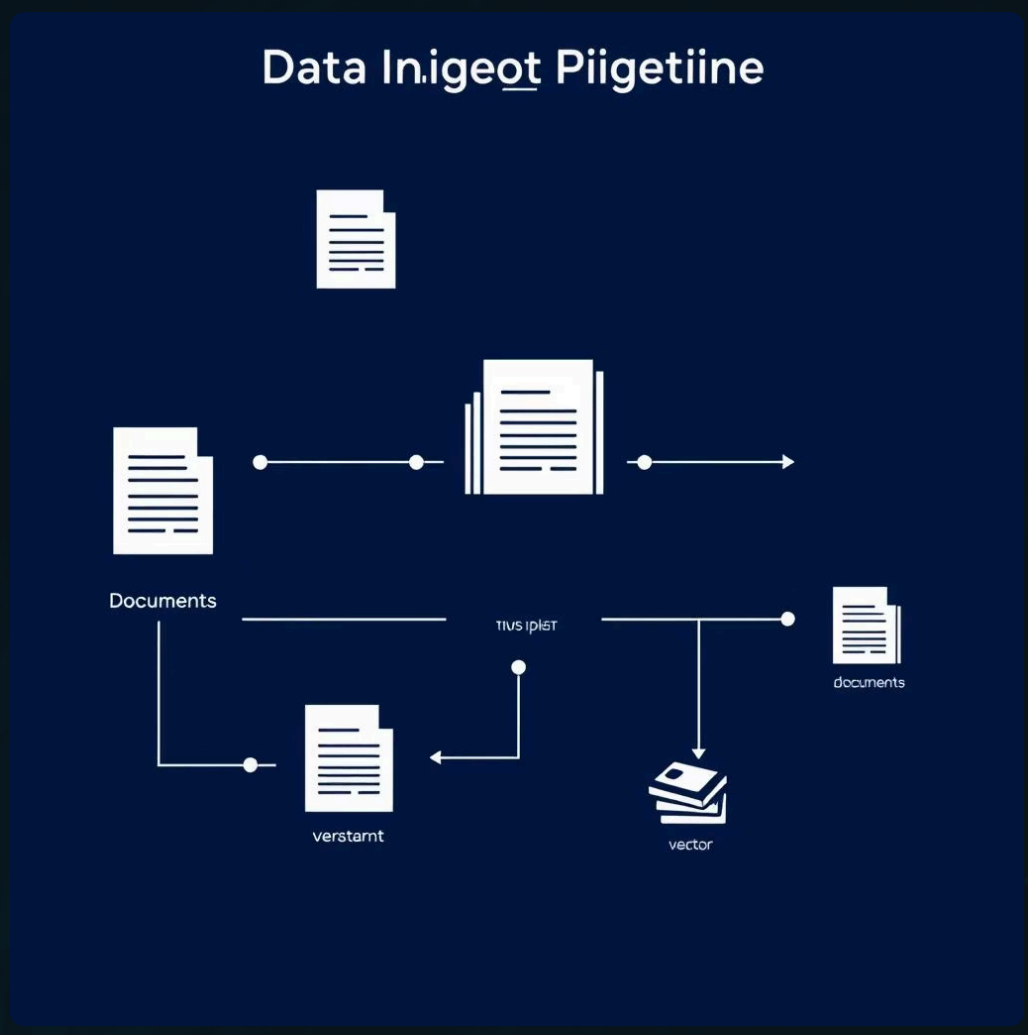
```
text_splitter.split_documents(docs)
```

3. Embedding Generation

Each text chunk is transformed into a high-dimensional vector (embedding) using an embedding model. These embeddings capture the semantic meaning of the text, enabling similarity search.

Example:

```
OpenAIEmbeddings()
```



Building the Retrieval System with Vector Stores

Vector stores are databases designed to efficiently store and query vector embeddings. They are crucial for quickly finding relevant document chunks based on semantic similarity to a user's query.

Embedding Comparison

When a query comes in, it's also embedded. The vector store compares the query embedding with all stored document embeddings.

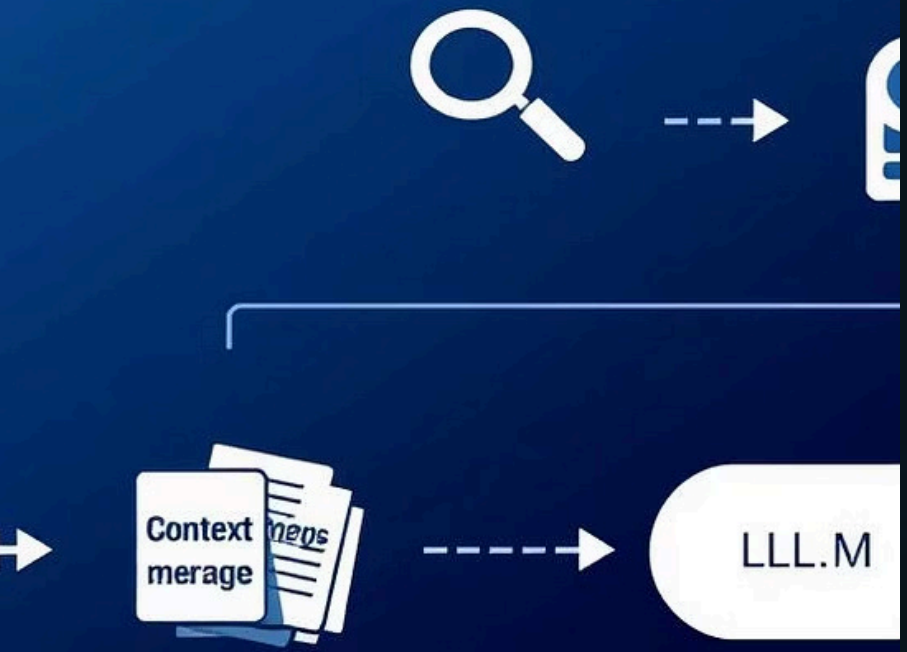
Similarity Search

Using algorithms like cosine similarity, the vector store identifies document chunks that are most semantically similar to the query.

Retrieval

The top-k most similar document chunks are retrieved and passed to the LLM as context.

Popular vector store options include FAISS, Chroma, Pinecone, and Weaviate, each offering different features and scalability.



Integrating Retrieval with Language Models

Once relevant context is retrieved, LangChain's chains combine this information with the user's query to form a comprehensive prompt for the LLM. This ensures the LLM generates context-aware and accurate responses.

LangChain Components in Action:

- **Retrievers:**
- **Chains (e.g., StuffDocumentsChain):**
- LLMs:

Key Takeaways & Next Steps

Key Takeaways

- **RAG Enhances LLMs:**
- **LangChain Simplifies Development:**
- **Data Preparation is Crucial:**
- **Vector Stores Enable Efficient Retrieval:**

Next Steps

- **Experiment:**
- **Explore Vector Stores:**
- **Evaluate Performance:**
- **Advanced Techniques:**

Thank you! Questions?