Retrieval-Augmented Generation (RAG): Extending LLM Capabilities with External Data

A workshop for software engineering interns

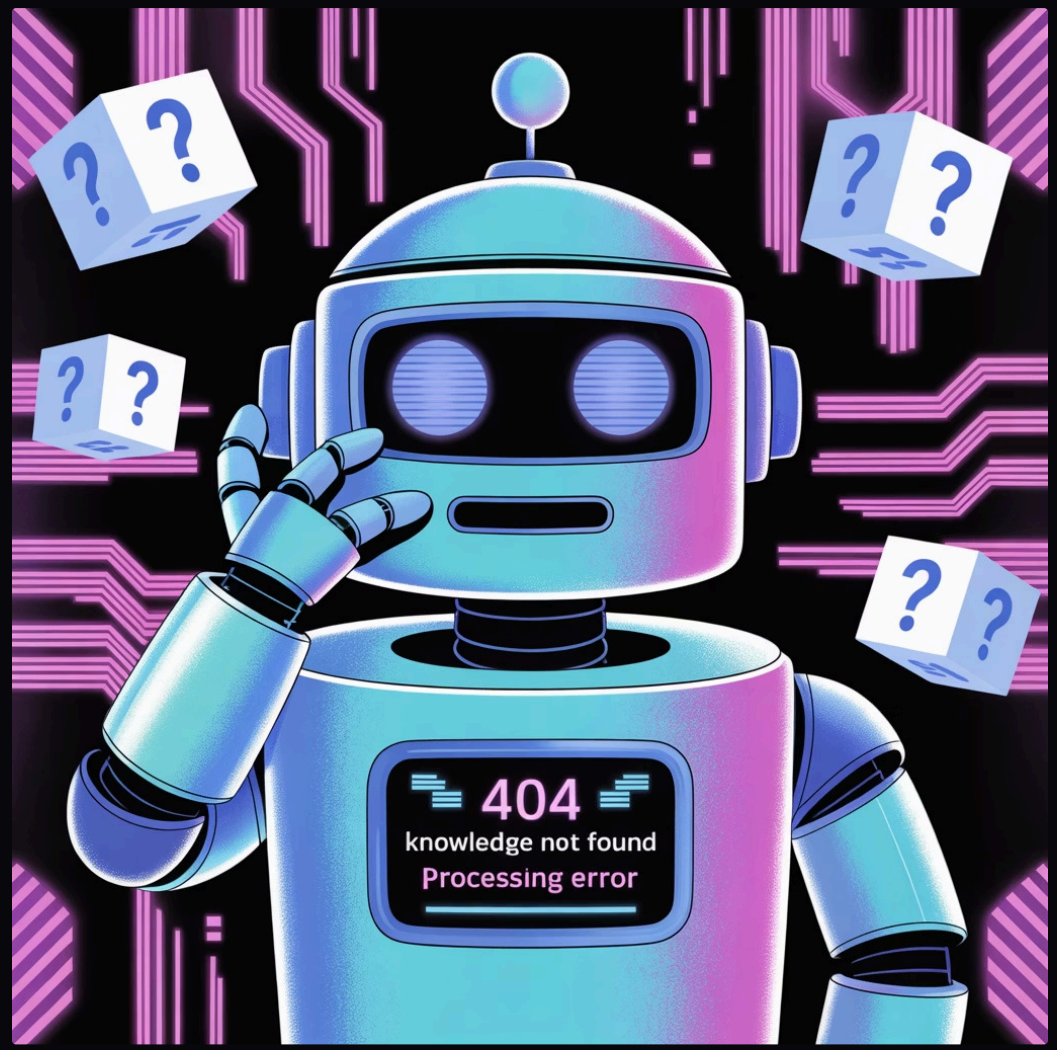
The Limitations of Standard LLMs

Large Language Models are trained on vast amounts of data, but that data is:

- Static (fixed at training time)
- Often outdated (trained months or years ago)
- Limited to public information
- Not specific to your organization

This creates significant blind spots when users ask about:

- Real-time information
- Private or proprietary data
- Recent events after training cutoff



Questions LLMs Can't Answer Without External Data

1

Specific User Data

"What's the exact status of my shipment order #12345?"

The model has no access to your specific order database or tracking systems.

2

Private Documents

"Can you summarize the latest internal company report?"

The model has never seen your internal documents unless specifically provided.

3

Real-time Information

"How many open support tickets do we have right now?"

The model can't access your ticketing system or real-time operational data.

How can we give our LLM access to external data it doesn't know?

Real-Life Scenarios that Require External Data

- **Customer Support Chatbots**

  Accessing real-time order statuses, product information, and company-specific FAQs to provide accurate assistance.

- **Enterprise Knowledge Bases**

  Searching through internal documentation, policies, and regulations to answer employee questions.

- **Medical Research Assistants**

  Retrieving current medical research papers and clinical guidelines to support healthcare professionals.

- **Legal Advisory Systems**

  Accessing current law documents, precedents, and specific court cases to provide legal insights.

We need a method to retrieve external data dynamically when the model needs it.

Introducing Retrieval-Augmented Generation (RAG)

> Retrieval-Augmented Generation (RAG) combines Large Language Models with external data retrieval systems to enhance their ability to answer questions accurately by dynamically fetching relevant information.

The Key Idea: Retrieve → Inject → Generate

RAG enables LLMs to answer questions based on information they were never trained on, creating more accurate, up-to-date, and contextually relevant responses.

How RAG Works: End-to-End Process

Data Preparation

Load external data (documents, websites, databases)

Split into manageable chunks

Convert chunks into vector embeddings

Store in vector database

Retrieval Process

Receive user question

Convert question to vector embedding

Search vector database for similar content

Retrieve most relevant chunks

Generation Process

Inject retrieved chunks into prompt

Send enhanced prompt to LLM

LLM generates response based on provided context

Return grounded, accurate answer to user

Step 1: Loading and Chunking Data

What It Means

- Loading documents from various sources (PDFs, web pages, databases)
- Splitting content into smaller, manageable chunks
- Preserving semantic meaning while creating appropriately sized units

Why It Matters

- Enables granular retrieval of precise, relevant information
- Optimizes for vector database storage and retrieval performance
- Balances context preservation with retrieval precision

LangChain Implementation

```
from langchain.document_loaders import TextLoader, PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Load document
loader = PyPDFLoader("company_handbook.pdf")
documents = loader.load()

# Split into chunks
text_splitter = RecursiveCharacterTextSplitter(
 chunk_size=500,
 chunk_overlap=50
)
chunks = text_splitter.split_documents(documents)
```

Step 2: Embedding Data

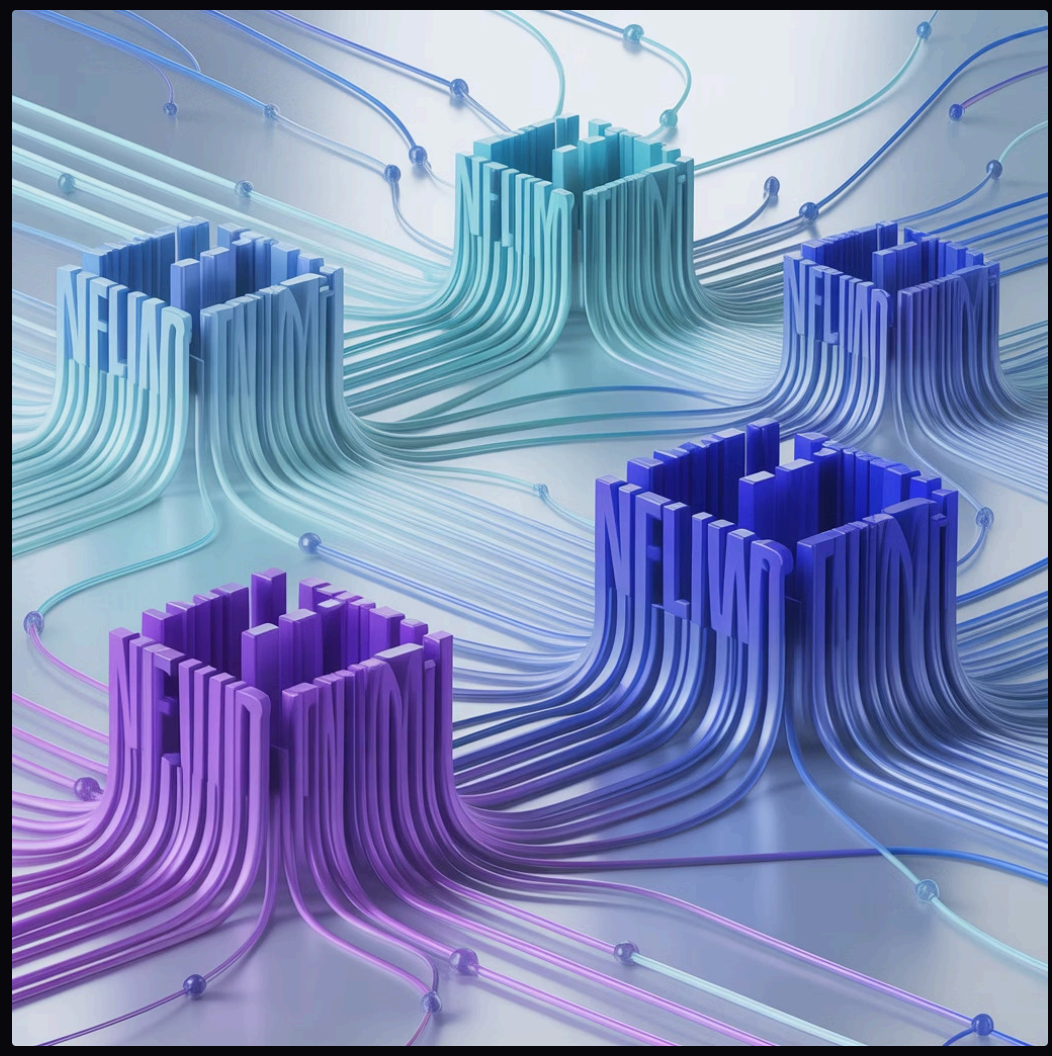
What It Means

Transforming text chunks into numeric vectors (embeddings) that capture semantic meaning

> ⓘ Embeddings are high-dimensional vectors (e.g., 1536 dimensions for OpenAI embeddings) that represent the semantic content of text.

Why It Matters

- Enables similarity search based on meaning, not just keywords
- Handles synonyms, different languages, and different phrasings
- Converts text into a format that computers can efficiently process



LangChain Implementation

```
from langchain.embeddings import OpenAIEmbeddings

# Initialize embeddings model
embeddings = OpenAIEmbeddings()

# Generate embeddings for each chunk
embedded_chunks = embeddings.embed_documents(
 [chunk.page_content for chunk in chunks]
)
```

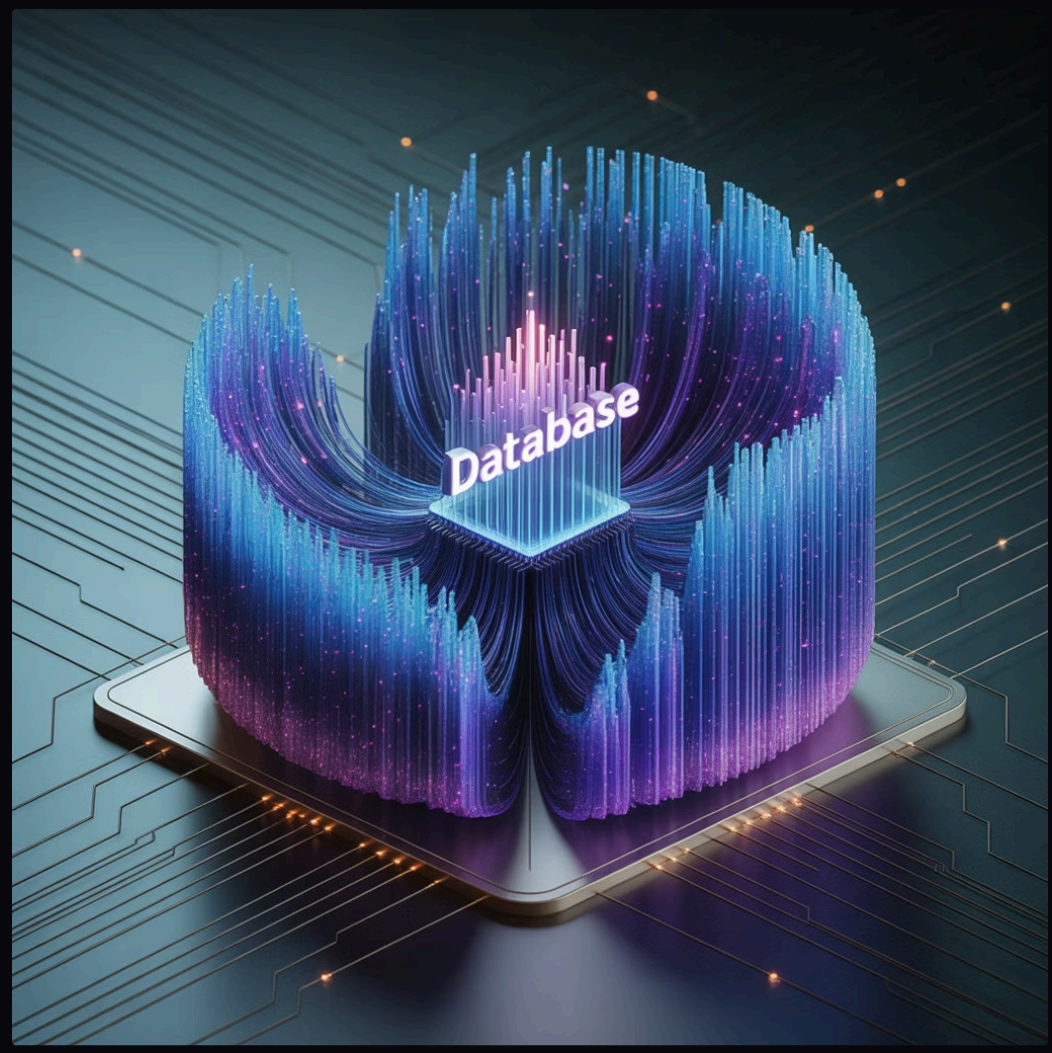
Step 3: Storing Embeddings in Vector Store

What It Means

Storing vector embeddings in specialized databases optimized for fast similarity searches

Common Vector Databases

- **Elasticsearch:** Highly scalable, enterprise-grade search engine
- **FAISS:** Facebook AI's efficient similarity search library
- **Chroma:** Open-source embedding database for AI applications
- **Pinecone:** Managed vector database service
- **Weaviate:** Knowledge graph and vector search engine



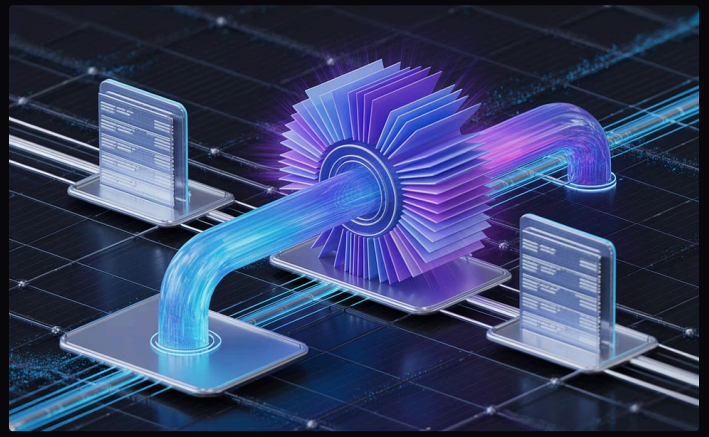
LangChain Implementation

```
from langchain.vectorstores import FAISS

# Create vector store
vectorstore = FAISS.from_documents(
 documents=chunks,
 embedding=embeddings
)

# Save for future use
vectorstore.save_local("company_faiss_index")
```

Step 4: Retrieval via Similarity Search

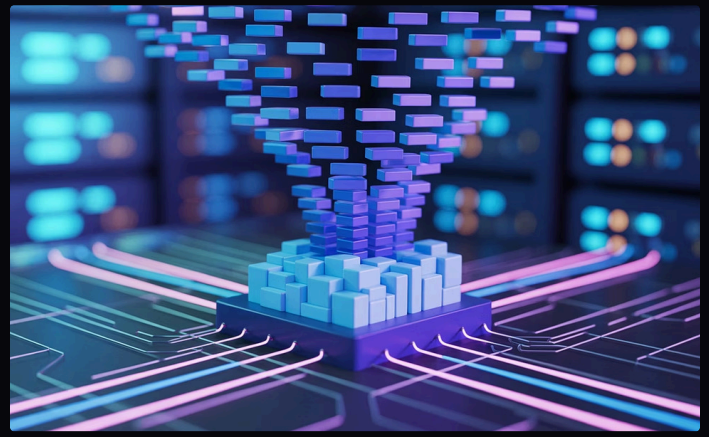


Convert Query to Vector

The user's question is transformed into the same vector space as the stored documents using the same embedding model.

Calculate Similarity

Vector database calculates similarity scores (often using cosine similarity) between query vector and stored document vectors.



Retrieve Top Matches

The most similar documents (highest similarity scores) are retrieved and prepared for inclusion in the LLM prompt.

```
# LangChain Implementation
relevant_docs = vectorstore.similarity_search(
 query="What is our company's remote work policy?",
 k=3 # Retrieve top 3 most relevant chunks
)
```

Step 5: Generation (Prompt Injection)

What It Means

Injecting retrieved context into a carefully crafted prompt for the LLM

Why It Matters

- Ensures the LLM answers based on the retrieved external data
- Provides specific, relevant context for the question
- Reduces hallucination by grounding responses in facts
- Maintains conversation history for multi-turn interactions

LangChain Implementation

```
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

# Initialize LLM
llm = ChatOpenAI(model="gpt-3.5-turbo")

# Create RAG chain
qa_chain = RetrievalQA.from_chain_type(
 llm=llm,
 chain_type="stuff", # Simple method that puts all documents in
context
 retriever=vectorstore.as_retriever()
)

# Get answer
response = qa_chain.run("What is our company's remote work
policy?")
```

Deep Dive: Understanding Embeddings

What Are Embeddings?

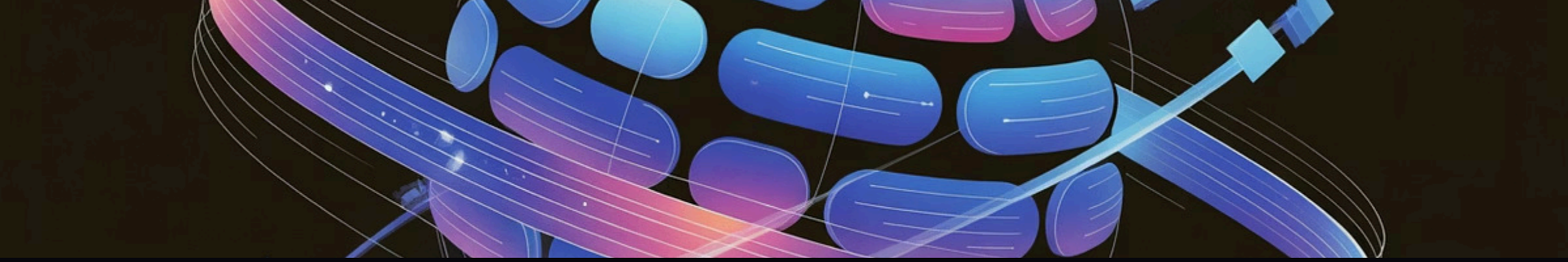
Embeddings are dense vector representations of text that capture semantic meaning in a multi-dimensional space.

Key Properties

- **Semantic Similarity:** Words/phrases with similar meanings cluster together
- **Dimensionality:** Typically 768-1536 dimensions depending on model
- **Language Agnostic:** Same concepts in different languages have similar vectors

Practical Applications

- Handling synonyms automatically (car/automobile)
- Understanding semantic relationships beyond keywords
- Enabling multilingual retrieval without explicit translation
- Finding conceptually related content even with different terminology



Vector Store Deep Dive: ElasticSearch

What Is ElasticSearch?

A distributed, RESTful search and analytics engine built on Apache Lucene, now with dedicated vector search capabilities.

Highly scalable from small projects to enterprise deployments handling billions of documents.

Advantages for RAG

- Combines traditional keyword search with vector similarity
- Real-time indexing for continuously updated data
- Robust filtering capabilities alongside vector search
- Enterprise-grade security and access controls
- Mature ecosystem with extensive monitoring tools

Implementation

```
from langchain.vectorstores import ElasticVectorSearch

elastic_vectorstore = ElasticVectorSearch(

elasticsearch_url="http://localhost:9200",
 index_name="company_docs",
 embedding=embeddings
)

elastic_vectorstore.add_documents(chunks)
```

Best Practices & Common Pitfalls

Best Practices

- **Chunking Strategy:** Optimal chunk size balances context and precision
- **Retrieval Settings:** Tune k (number of retrieved chunks) based on question complexity
- **Prompt Engineering:** Clear instructions to the LLM on how to use retrieved context
- **Metadata Filtering:** Use document metadata to narrow search scope
- **Reranking:** Apply secondary ranking to initial search results for higher relevance
- **Regular Updates:** Refresh your vector store as source documents change

Common Pitfalls

- **Chunk Size Issues:** Too large → irrelevant info; Too small → fragmented context
- **Poor Embeddings Quality:** Using inappropriate embedding models for your domain
- **Retrieval Without Context:** Not considering conversation history in multi-turn interactions
- **Over-retrieval:** Retrieving too much content can overwhelm context windows
- **Citation Neglect:** Not tracking which sources provided which information

Next Steps: Hands-On Workshop

What We'll Build

A simple RAG application using Python, LangChain, and ElasticSearch that can answer questions from our company documentation.

Workshop Format

- Step-by-step implementation with live coding
- Interactive debugging and troubleshooting
- Testing with real-world queries
- Discussion of optimization techniques

Prerequisites

- Python 3.8+ installed
- Basic familiarity with Python and pip
- Docker for running ElasticSearch (or use cloud service)
- OpenAI API key (provided for workshop)

Let's build a practical RAG system together!