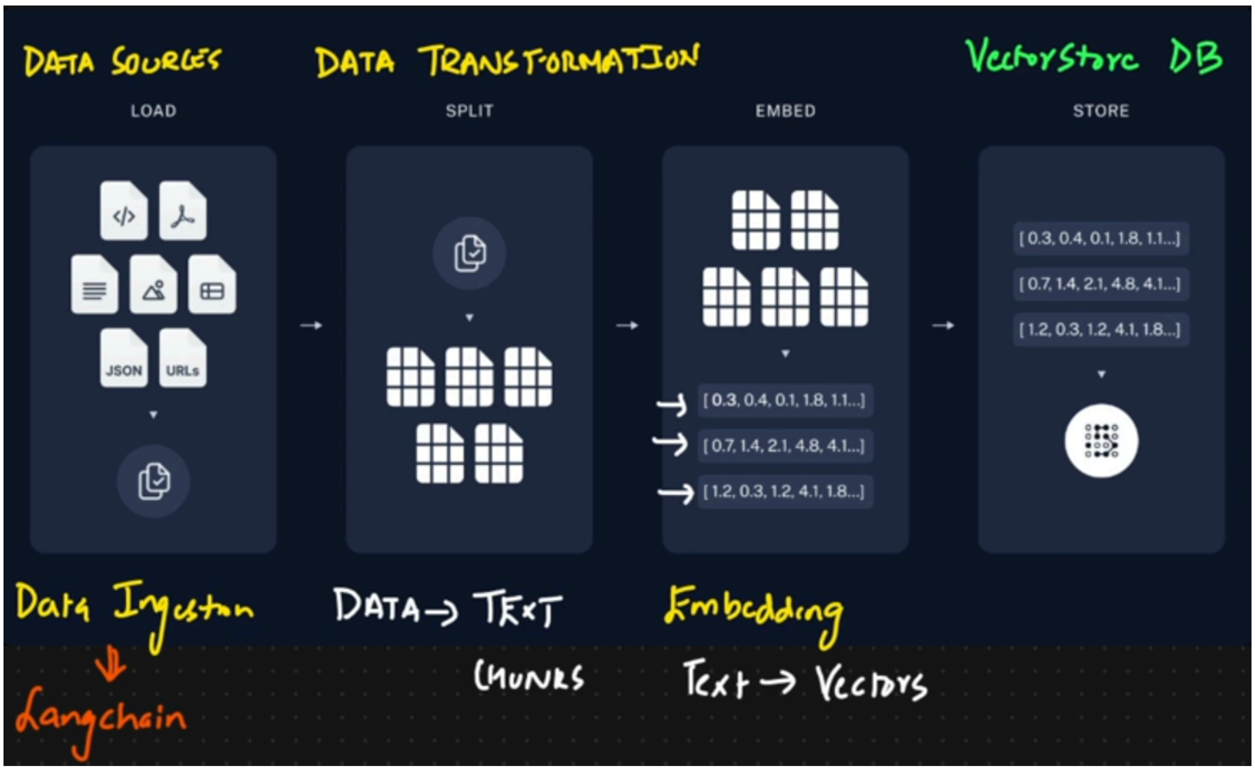
Components of LangChain

RAG (Retrieval-Augmented Generation) pipeline components



You want to **chat with your documents** (PDFs, Word, CSV, etc.), or search over them smartly using a Language Model (LLM).

To do that, the computer must:

1. Understand the document
2. Store it in a smart way
3. Search intelligently inside it

What is RAG?

RAG = Google Search + ChatGPT

- It's a technique where an AI model (**Generation**) fetches (**Retrieves**) relevant information from your documents *before* answering a question.
- Combines **search** (like Google) with **text generation** (like ChatGPT).

Why Use RAG?

- LLMs (like GPT) **don't know everything**—they're trained on old/public data.
- RAG lets you **"feed" custom data** (PDFs, websites, etc.) to the AI for accurate, up-to-date answers.

Real-World Example

Question: *"What's our company's refund policy?"*

- **Without RAG**: LLM guesses based on general knowledge.
- **With RAG**:
  1. Searches your internal **policy.docx** file.
  2. Finds the exact refund policy section.
  3. Generates an answer **quoting your actual policy**.

Key Benefits

✅ **No retraining needed** – Just add new files!

✅ **Reduces AI mistakes** – Grounded in your data.

✅ **Works with private data** – PDFs, emails, database

🥇 1. Load/Data Ingestion

This means **reading your document** (like a PDF, CSV, etc.) and bringing it into Python as text.

What: Import data from files/websites/databases into LangChain.

Like: A librarian gathering books for your research.

Tools:

- `WebBaseLoader`: Scrape websites
- `PyPDFLoader`: Read PDFs
- `DirectoryLoader`: Load all files in a folder

📚 Example:

```
from langchain_community.document_loaders import PyPDFLoader
loader = PyPDFLoader("myfile.pdf")
docs = loader.load()
```

- `docs` now contains the whole document as **strings of text**.
- If the PDF has 10 pages, each page may become one document.

🥈 2. Split

Large documents are **too long for LLMs to handle in one go**, so we **break them into smaller pieces** (called "chunks").

🔪 Why?

LLMs like GPT or DeepSeek have limits (e.g., 4096 tokens). You must keep chunks small enough to fit.

Popular Splitters:

- `RecursiveCharacterTextSplitter`: Generic purpose
- `MarkdownHeaderTextSplitter`: Preserves document structure

📚 Example:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
```

```
chunks = splitter.split_documents(docs)
```

Now each chunk is ~500 characters (or tokens), and they slightly overlap for context.

🥉 3. Embed

Now, we convert **text into → vectors (numbers)** — like turning sentences into coordinates in space.

This is called **embedding**. It's like giving meaning to text in math language.

🧠 Why?

LLMs and search tools **understand text better when it's a vector** — so you can find "similar meaning" not just same words.

Popular Embedders:

- `OpenAIEmbeddings` (paid)
- `HuggingFaceEmbeddings` (free, e.g., "all-MiniLM-L6-v2")

📚 Example:

```
from langchain.embeddings import HuggingFaceEmbeddings
embedding_model = HuggingFaceEmbeddings()
```

- Each chunk of text becomes a **vector (list of numbers like `[0.12, -0.98, 3.4, ...]`)**.

🥇 4. Store (Vector Store)

Now, we **store those embeddings (vectors)** in a **special database** called a **Vector Store**.

What:

Databases optimized for storing/querying embeddings.

Why:

Fast similarity searches over large datasets.

Popular options:

Name	Local?	Good for beginners?	**Example Code**
FAISS	✅ Yes	✅ Yes (offline)	`FAISS.from_documents(chunks, embedder)`
ChromaDB	✅ Yes	✅ Yes (easy API)	`Chroma.from_documents(chunks, embedder, persist_dir="./db")`
Pinecone	❌ No	❌ Needs API key	

📚 FAISS Example:

```
from langchain.vectorstores import FAISS
db = FAISS.from_documents(chunks, embedding_model)
```

Now your document is stored in a smart way — you can **search by meaning**.

🧭 5. Search / Retrieve

Later, when you ask a question, you **don't send everything to the LLM**. Instead:

1. Your question is also **embedded into a vector**
2. LangChain compares your question's vector with the document vectors
3. It returns **the most similar chunks**

This is called **retrieval** or **semantic search**.

What: Find relevant chunks for a query.

Like: Using a book's index to find relevant pages.

📚 Example:

```
retriever = db.as_retriever()
results = retriever.invoke("Explain LangChain")
```

You now get only the **most relevant pieces** of your document, which are then passed to the LLM to answer.

```
query = "What's LangChain?"
similar_chunks = vector_db.similarity_search(query, k=3)  # Top 3 matches
```

Full RAG Pipeline Visualization:

```
[PDF/Website]
    → Load → Split → Embed → Store (FAISS/Chroma)
                           ↓
                 Query → Retrieve → [LLM] → Answer
```

✅ Summary in One Line

Term	Meaning
Load	Read the document (PDF, CSV, etc.)
Split	Break into small chunks
Embed	Turn text into numbers (vectors)
Store	Save vectors in a database (FAISS, Chroma, etc.)
Retrieve	Find similar chunks based on a question

🧠 Trivia for Better Understanding

- **Embedding** = like turning every sentence into a **GPS coordinate**
- **Vector Store** = like a **map** where every sentence has a location

- **Retrieval** = find **nearest neighbors** (sentences close in meaning)

Complete Example:

```
# 1. Load
loader = PyPDFLoader("report.pdf")
docs = loader.load()

# 2. Split
splitter = RecursiveCharacterTextSplitter(chunk_size=1000)
chunks = splitter.split_documents(docs)

# 3. Embed + Store
embedder = HuggingFaceEmbeddings()
vector_db = FAISS.from_documents(chunks, embedder)

# 4. Retrieve
query = "Summarize key findings"
results = vector_db.similarity_search(query)

# 5. Generate answer
from langchain.chains import RetrievalQA
qa_chain = RetrievalQA.from_chain_type(llm, retriever=vector_db.as_retriever
())
print(qa_chain.run(query))
```

Key Concepts

- **Chunk Size**: Typically 500-1500 characters (balance context vs. precision)
- **Embedding Models**: Smaller ones (e.g., 384-dim) work well for most cases
- **Vector DB Choice**: FAISS for quick tests, Chroma for local persistence, Pinecone for production