
Naresh Edagotti
@Statfusionai

RAG with LangChain

A Comprehensive Guide to Retrieval-Augmented Generation

What is LangChain?

LangChain is a framework for building LLM-powered applications with:

- **Modular Components:** Pre-built modules for document loading, embeddings, chains
- **Easy Integration:** Works with popular LLMs, vector databases, and tools
- **Production Ready:** Built for scalable applications

What We Need to Build RAG

Installation:

```
pip install langchain_community pypdf langchain
pip install langchain_huggingface faiss-cpu langchain_groq
```

Core Components:

1. **Document Loaders** → Load data from various sources
2. **Text Splitters** → Break documents into chunks
3. **Embeddings** → Convert text to vectors
4. **Vector Stores** → Store and search embeddings
5. **Retrievers** → Find relevant documents
6. **LLMs** → Generate responses
7. **Evaluation** → Assess performance

Document Loading

What it does:

Converts various document formats into structured text that can be processed.

Available Loaders:

- PyPDFLoader: PDF files
- TextLoader: Plain text files
- WebBaseLoader: Website content
- CSVLoader: CSV files
- WikipediaLoader: Wikipedia articles

Implementation:

```python
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("/content/document.pdf")
documents = loader.load()
print(f"Loaded {len(documents)} documents")
```

Preprocessing(Manual Required)

Important:

LangChain doesn't provide built-in preprocessing. You must write custom code.

When to preprocess:

- Documents have noise, formatting issues
- Need to clean headers, footers, page numbers
- Want to standardize text format

Example Implementation:

```python
import re

def preprocess_text(text):
    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text)
    # Remove page numbers
    text = re.sub(r'Page \d+', '', text)
    # Remove special characters
    text = re.sub(r'[^\w\s\.\,\!\?]', '', text)
    return text.strip()

# Apply to documents
for doc in documents:
    doc.page_content = preprocess_text(doc.page_content)
```

Text Chunking

What it does:

Splits large documents into smaller pieces for better retrieval and processing.

Why needed:

- Embedding models have token limits (512-1024 tokens)
- Smaller chunks = more precise retrieval
- Maintains context with overlapping chunks

Implementation:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=300,      # Max characters per chunk
    chunk_overlap=50     # Overlap between chunks
)
docs = text_splitter.split_documents(documents)
print(f"Split into {len(docs)} chunks")
```

Key Parameters:

- chunk_size: 200-1000 characters (300 is optimal)
- chunk_overlap: 10-20% of chunk_size
- separators: ["\n\n", "\n", ".", " "] (paragraph → sentence → word)

Embeddings

What it does:

Converts text chunks into numerical vectors that capture semantic meaning.

Popular Models:

- all-mpnet-base-v2: Best balance of quality and speed
- all-MiniLM-L6-v2: Fastest, good for large datasets
- text-embedding-ada-002: OpenAI's high-quality model

Implementation:

```python
from langchain_huggingface import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-mpnet-base-v2",
    model_kwargs={'device': 'cpu'}
)
```

Vector Database

What it does:

Stores embeddings and enables fast similarity search to find relevant chunks.

Why FAISS:

- Fast: Optimized for similarity search
- Scalable: Handles millions of vectors
- Free: Open-source Facebook AI tool
- GPU Support: Faster processing with CUDA

Implementation:

```python
from langchain_community.vectorstores import FAISS

# Create vector store from documents
db = FAISS.from_documents(docs, embeddings)
print("✅ Vector store created")
```

Alternatives:

- Pinecone: Managed cloud service
- Chroma: Simple, lightweight
- Qdrant: High-performance option

Retrieval

What it does:

Finds the most relevant document chunks based on user query similarity.

How it works:

- Convert query to embedding
- Search vector database for similar chunks
- Return top-k most relevant results

Implementation:

```python
# Create retriever
retriever = db.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 5}  # Return top 5 chunks
)

# Retrieve documents
query = "What is RAG?"
retrieved_docs = retriever.get_relevant_documents(query)
print(f"Retrieved {len(retrieved_docs)} documents")
```

Key Settings:

- k: Number of chunks to retrieve (3-10)
- search_type: "similarity" or "mmr" (for diversity)
- score_threshold: Minimum similarity score

Generation

What it does:

Uses retrieved context to generate accurate, contextual responses with an LLM.

How it works:

- Combines retrieved chunks with user query
- Creates structured prompt with context
- LLM generates response based on provided context

Chain Types:

- stuff: Fast, limited context
- map_reduce: Handles more context
- refine: Most thorough but slow

Generation

Implementation:

```
from langchain_groq import ChatGroq
from langchain.chains import RetrievalQA

GROQ_API_KEY="your_api_key"

# Initialize LLM
llm = ChatGroq(
    temperature=0,                  # Deterministic responses
    model_name="gemma2-9b-it",
    api_key=GROQ_API_KEY
)

# Create RAG chain
rag_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",             # Combine all context in one prompt
    retriever=retriever
)

# Generate response
response = rag_chain.invoke({"query": query})
print(response["result"])
```

Evaluation

What it does:

Measures RAG system performance to ensure quality and identify improvements.

Key Metrics:

- Faithfulness: Answer stays true to source documents
- Relevance: Answer addresses the question
- Retrieval Quality: Retrieved chunks are relevant

Implementation:

```python
from langchain.evaluation.qa import QAEvalChain

# Prepare evaluation data
examples = [{"query": query, "answer": "Ground truth answer"}]
predictions = [{"query": query, "result": response["result"]}]

# Evaluate
eval_chain = QAEvalChain.from_llm(llm)
graded_outputs = eval_chain.evaluate(examples, predictions)
print(graded_outputs)
```

Evaluation Types:

- Automated: BLEU, ROUGE, BERTScore
- Human: User ratings, comparative analysis
- Continuous: A/B testing, feedback loops

Key Success Tips

Optimization:

- Chunk Size: Start with 300 characters, adjust based on your data
- Overlap: Use 50-100 characters for context preservation
- Retrieval: Experiment with k=3 to k=10 based on query complexity
- Temperature: Keep at 0 for factual responses

Common Issues:

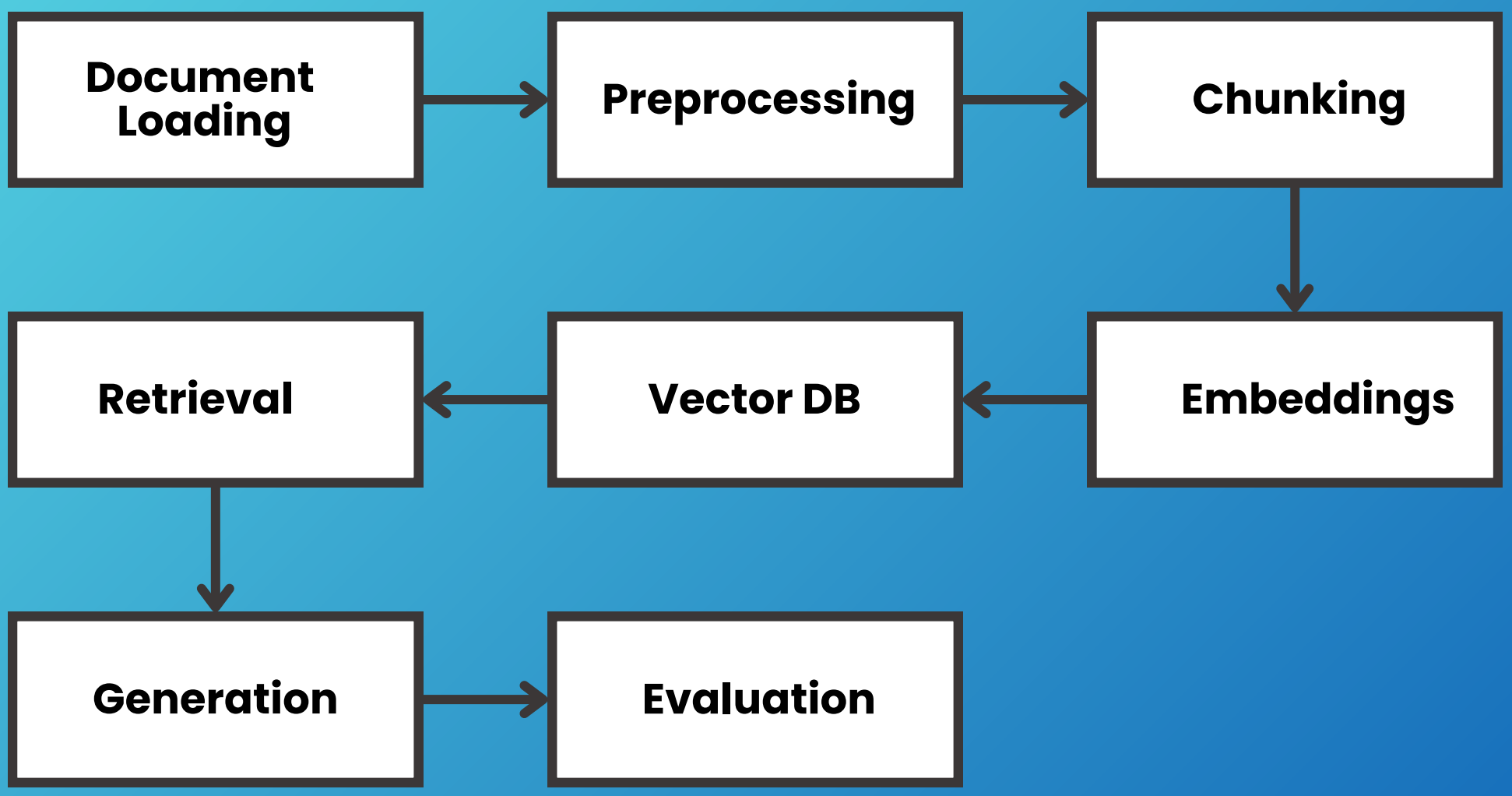
- Poor Retrieval: Check chunk size and embedding model
- Hallucination: Ensure context is relevant and sufficient
- Slow Response: Reduce chunk size or number of retrieved docs

Production Ready:

- Caching: Store frequently accessed embeddings
- Monitoring: Track query performance and user feedback
- Error Handling: Graceful failures and fallbacks
- Security: Protect API keys and sensitive data

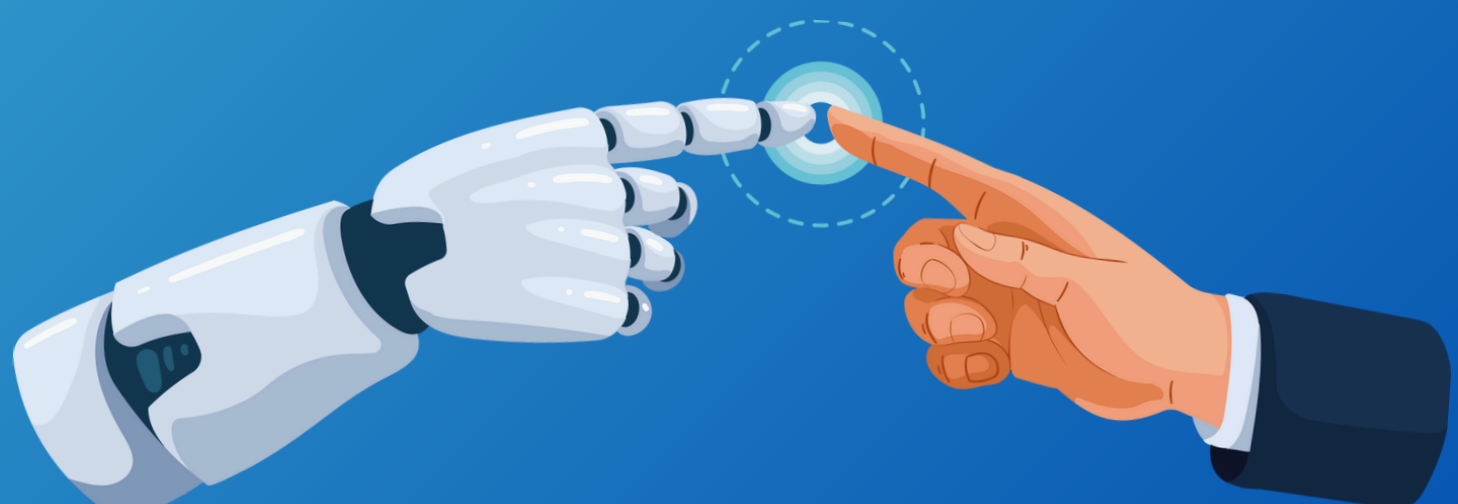
Summary

RAG Pipeline:



Remember:

- Preprocessing requires manual implementation
- Chunk size affects retrieval quality
- Evaluation is crucial for production systems
- Start simple, then optimize based on results



LIKE THIS CONTENT?

FOLLOW FOR MORE!

NARESH EDAGOTTI

LIKE REPOST SAVE