Tajamul Khan

RAG Cheat Sheet

@Tajamulkhann

LangChain

LangChain is a framework designed for building language model (LLM)-powered applications with modular, production-ready components.

RAG

Retrieval-Augmented Generation enhances language model outputs by fetching relevant, up-to-date information from external knowledge sources and combining it with generative AI capability.

Installation

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

@Tajamulkhann
@Tajamul.datascientist

Components

Document Loaders → Bring in data from different sources.

Text Splitters → Break documents into smaller, manageable chunks.

Embeddings → Turn text into numerical vectors for understanding.

Vector Stores → Save and efficiently search these vectors.

Retrievers → Find the most relevant chunks for a query.

Generators → Generate answers or insights using the retrieved information.

Evaluation → Check how accurate or useful the results are.

@Tajamulkhann
@Tajamul.datascientist

Preprocessing

LangChain doesn't include built-in preprocessing—you need to handle it yourself.
Use: Documents often have noise and formatting like headers, footers, or page numbers, so cleaning and standardizing text is essential.

```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)
```


@Tajamulkhann
@Tajamul.datascientist

Document Loaders

Import data from multiple sources.

PDF

```
from langchaincommunity.documentloaders import PyPDFLoader
loader = PyPDFLoader("file.pdf")
documents = loader.load()
```

Text

```
from langchaincommunity.documentloaders import TextLoader
loader = TextLoader("file.txt")
documents = loader.load()
```

Websites

```
from langchain_community.document_loaders import WebBaseLoader
loader = WebBaseLoader("https://example.com")
documents = loader.load()
```

CSV

```
from langchain_community.document_loaders import CSVLoader
loader = CSVLoader("file.csv")
documents = loader.load()
```


@Tajamulkhann
@Tajamul.datascientist

Text Splitters

Break large documents into manageable, context-preserving **chunks.**

Best Practice: Use 300 character chunks with 50 character overlap.

Recursive Character Text Splitter

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

text = """
LangChain is a powerful framework for building applications with language models.
It provides document loaders, text splitters, embeddings, vector stores, and more.
You can use it for RAG, chatbots, QA systems, and custom LLM workflows.
"""

splitter = RecursiveCharacterTextSplitter(chunk_size=50, chunk_overlap=10)
chunks = splitter.split_text(text)
print(chunks)
# Output:
# ['LangChain is a powerful framework for building ',
#  'for building applications with language models.\nIt ',
#  'models.\nIt provides document loaders, text ',
#  'splitters, embeddings, vector stores, and more.\nYou ',
#  'You can use it for RAG, chatbots, QA systems, ',
#  'QA systems, and custom LLM workflows.']
```


@Tajamulkhann
@Tajamul.datascientist

Embeddings

Convert text chunks to semantic vectors.

Common Models:

- HuggingFace ('all-mpnet-base-v2', 'all-MiniLM-L6-v2')
- OpenAI ('text-embedding-ada-002')

Hugging Face Embedding

```
from langchain_huggingface import HuggingFaceEmbeddings

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

Open AI Embedding

```
from langchain_openai import OpenAIEmbeddings
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")
```


@Tajamulkhann
@Tajamul.datascientist

Vector Stores

Stores embeddings and enables fast similarity search (usually using cosine similarity) to find relevant chunks.

Popular:

- FAISS for fast local search
- Chroma/Pinecone for scalable options.

FAISS

```
from langchaincommunity.vectorstores import FAISS
db = FAISS.from_documents(docs, embeddings)
print("Vector store created")
```

@Tajamulkhann
@Tajamul.datascientist

Retrievers

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

```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: Min. similarity score

@Tajamulkhann

@Tajamul.datascientist

Generators

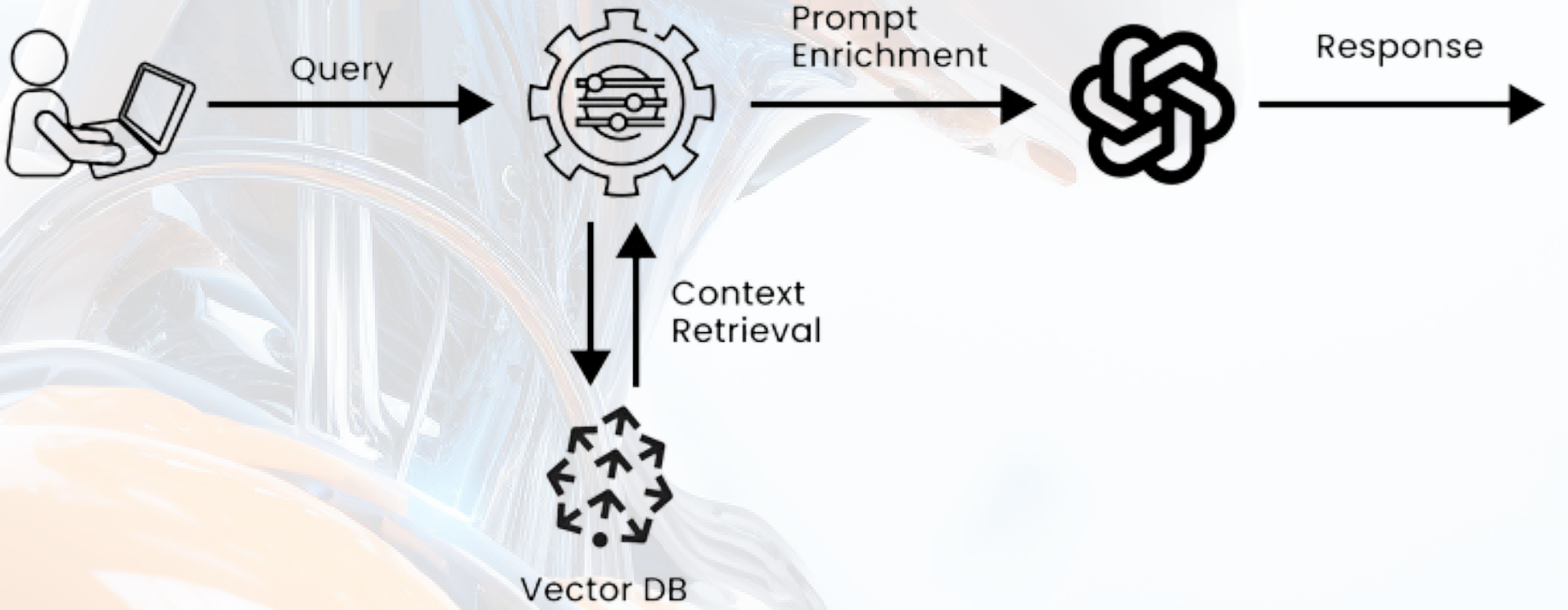
retrieval and generation—process user query, fetch relevant info, and generate an informed response with LLM.

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-4", temperature=0)
```

Chain Types:

- stuff: fast, for small context
- map-reduce: for longer docs
- refine: slowest, highest quality

```
from langchain.chains import RetrievalQA
rag_chain = RetrievalQA.from_chain_type(llm=llm, retriever=re
```

@Tajamulkhann
@Tajamul.datascientist

Evaluation

Evaluates RAG system performance to maintain quality and highlight areas for improvement.

Key Metrics:

- Faithfulness → Response accurately reflects the source documents.
- Relevance → Response directly answers the question.
- Retrieval Quality → Retrieved chunks are useful and on-topic.

```
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

@Tajamulkhann
@Tajamul.datascientist

RAG Tips

Success Tips for RAG

- Chunking → ~300 chars + 50–100 overlap.
- Retrieval → Try k=3–10 based on query.
- LLM → Temperature = 0 for factual output.

Common Issues

- Poor retrieval → adjust chunk size/embeddings.
- Hallucination → improve context quality.
- Slow response → reduce chunk size or k.

Production Ready

- Caching frequent embeddings.
- Monitor performance & feedback.
- Add fallbacks for errors.
- Secure API keys & data.

@Tajamulkhann
@Tajamul.datascientist

RAG Pipeline

Document Loading
Preprocessing
Chunking
Embeddings
Vector DB
Retrieval
Generation
Evaluation

@Tajamulkhann
@Tajamul.datascientist

Found Helpful ?

Repost

@Tajamulkhann

@Tajamul.datascientist

Follow for more!