ChatGPT

LangChain으로 살펴보는 Retrieval-Augmented Generation (RAG)

RAG란 무엇이며 왜 중요한가?

Retrieval-Augmented Generation (RAG)은 대형 언어 모델(LLM)의 응답 생성 과정에 외부 지식 정보를 결합하여 모델의 한계를 보완하는 기술입니다. 쉽게 말해, **LLM이 답변을 생성하기 전에 모델의 학습 데이터에 없는 최신 정보나 도메인 특화 지식을 검색하여 활용**하는 방법입니다 [1]. 이를 통해 거대한 사전 학습으로 얻은 언어 생성 능력에 **최신의 정확한 데이터**를 덧붙여, 더 신뢰성 있고 유용한 답변을 얻을 수 있습니다.

LLM만으로 질문에 답할 경우 몇 가지 문제가 발생할 수 있습니다. 예를 들어 **모델이 답을 모를 때 사실이 아닌 내용을 지어내거나 (일명 환각, hallucination)**, 학습 시점 이후의 최신 정보에 대해 **구 outdated된 정보를 자신 있게 말하거나**, 출처가 불분명한 정보를 생성하는 등의 한계가 있습니다 [2]. 이러한 한계 때문에 사용자 입장에서는 모델의 답변을 그대로 신뢰하기 어려울 수 있습니다. **RAG는 LLM의 이런 문제를 해결하는 한 방안으로 고안되었습니다**. 모델이 답변을 만들 때 **미리 정해둔 신뢰 가능한 데이터베이스에서 관련 정보를 검색하여 활용하도록 함으로써**, 응답의 정확성과 최신성을 높이고 사용자가 답변의 근거를 확인할 수 있게 도와줍니다 [3].

요약하면, **RAG는 LLM의 강력한 언어 생성 능력에 검색 엔진의 정보 탐색 능력을 결합**한 것으로, **최신의 신뢰할 수 있는 정보에 기반한 답변**을 생성하여 **환각을 줄이고 사용자 신뢰를 향상**시키는 기법입니다 [4]. 이러한 이유로 RAG는 **기업 내 사설 지식 베이스 질의응답**, **최신 뉴스나 논문 검색 기반 챗봇**, **제품 설명서 Q&A** 등 **LLM 응용 분야에서 정확성과 현행성을 높이는 핵심 기술**로 각광받고 있습니다.

RAG의 동작 원리 (요약)

RAG 시스템은 **데이터 준비 단계**와 **질의 응답 단계**로 나눌 수 있습니다. 전체적인 흐름은 다음과 같습니다 [5]:

1. **데이터 준비 (Ingestion)**: 우선 RAG에 사용할 외부 지식 데이터를 수집합니다. 예를 들어 사내 문서, 위키 자료, 논문, 매뉴얼 등을 **문서 형태로 로드**하고 **필요에 따라 전처리 및 청크 단위로 분할**합니다. 그런 다음 해당 **텍스트 조각들마다 임베딩(벡터 표현)을 생성하여 벡터 데이터베이스(vector store)에 저장**해 둡니다. 이 단계까지 완료하면 LLM이 참고할 수 있는 **"검색용 지식 베이스"**가 벡터 형태로 구축됩니다.

2. **검색 (Retrieval)**: 사용자가 **질문(query)**을 하면, **질문도 임베딩 벡터로 변환하여** 앞서 구축된 벡터 DB에서 **의미적으로 유사한 벡터들을 검색**합니다. 이를 통해 **질문과 관련성 높은 문서 조각들**을 찾아냅니다. 이 검색 작업을 수행하는 모듈을 **리트리버(retriever)**라고 부릅니다.

3. **프롬프트 보강 (Augmentation)**: 검색된 관련 문서 조각들을 **원문 내용(컨텍스트)**으로 삼아 사용자 질문과 합칩니다. 즉, **LLM에 줄 프롬프트(prompt)**에 사용자 질문뿐 아니라 **찾아온 참고 정보들을 함께 포함**시킵니다. 이렇게 프롬프트를 구성할 때는 보통 미리 정의된 **프롬프트 템플릿(prompt template)**을 사용하여 "다음 정보들을 기반으로 질문에 답하라"는 형식으로 맥락을 제공합니다 [6].

4. **응답 생성 (Generation)**: 최종적으로 **LLM이 보강된 프롬프트를 입력받아 답변을 생성**합니다. 프롬프트에는 질문 관련 핵심 정보들이 포함되어 있으므로, LLM은 자신이 원래 알고 있던 지식과 함께 **제공된 컨텍스트를 참고하여 더욱 정확하고 구체적인 답변**을 내놓을 수 있습니다 [7]. 이로써 사용자는 **근거 자료에 기반한 신뢰도 높은 답변**을 얻을 수 있고, LLM의 환각 현상은 줄어듭니다.

RAG 파이프라인의 일반적인 구조를 나타낸 그림입니다. **위쪽**은 데이터 적재 단계로, 원본 문서를 작은 청크로 분할한 뒤 임베딩하여 벡터 DB에 저장하는 과정을 보여줍니다. **가운데**는 사용자의 **질의**를 임베딩하여 벡터 DB에서 유사한 문서 조각을 검색(Retrieval)하고, 해당 조각들을 모아 **프롬프트에 추가하는 과정(Augmentation)**을 나타냅니다. **아래쪽**에서는 보강된 프롬프트를 LLM에 보내 응답을 생성(Generation)하는 단계를 보여줍니다. 이처럼 RAG 시스템은 **사전에 구축한 지식 정보**와 **실시간 사용자 질문**을 결합하여 응답을 생성하는 일련의 과정을 거칩니다.

RAG 구성 요소 및 핵심 개념

이제 RAG를 구성하는 주요 컴포넌트들과 핵심 개념들을 하나씩 살펴보겠습니다. 각 용어가 무엇을 의미하며 서로 어떻게 연결되는지 이해하는 것이 중요합니다.

- **문서 로더 (Document Loader)**: 원본 데이터를 불러와 **일관된 문서 객체 형태로 변환**하는 도구입니다. LangChain에서는 CSV, PDF, 웹 페이지, 데이터베이스 등 **다양한 형식의 데이터를 읽어서 내부** Document **객체로 변환**할 수 있는 수백 종의 로더를 제공합니다 [8]. Document 객체에는 본문 내용(`page_content`), ID, 출처 등 메타데이터가 포함되어 있어 후속 처리에 활용됩니다. 문서 로더를 사용하면 외부 데이터를 일일이 파싱할 필요 없이 쉽게 RAG 파이프라인에 넣을 수 있습니다.

- **문서 청크 분할 (Chunking)**: **긴 문서를 일정한 크기의 조각(chunk)들로 나누는 작업**입니다. 한 문서를 통째로 처리하면 길이가 너무 길어서 모델 입력 한계를 넘거나, 임베딩 품질이 떨어질 수 있습니다. 따라서 **문단이나 일정 토큰 수 단위로 문서를 쪼개어** 일관된 크기의 텍스트 블록들로 만들게 됩니다 [9]. 문서를 청크로 분할하면 **모델의 최대 입력 길이 제한에 대응**하고, **각 청크마다 더 집중된 주제 표현**이 가능해져 **검색 정확도**가 향상됩니다. LangChain은 `RecursiveCharacterTextSplitter` 등 다양한 **텍스트 분할 도구**를 제공하여, 문자 길이, 토큰 길이, 문단/문장 단위 등 여러 **전략으로 청크 분할**을 쉽게 할 수 있습니다 [10] [11].

- **임베딩 (Embedding)**: 텍스트를 숫자 벡터로 표현하는 방법입니다. 임베딩 모델(예: BERT, SBERT, OpenAI Embedding 등)은 단어나 문장 등을 **고차원 공간의 벡터로 변환하여, 의미적으로 유사한 내용은 벡터 공간에서 가깝게 위치**하도록 만듭니다 [12]. 예를 들어 "강아지"와 "고양이" 같은 단어는 임베딩 공간에서 가까운 좌표를 갖지만 "강아지"와 "테이블"은 거리가 멀게 나타납니다 [13]. RAG에서는 문서 청크와 사용자 쿼리 모두 임베딩으로 바꾸어 **의미 기반 유사도를 계산**하며, **키워드 매칭 이상의 semantic search**를 가능하게 합니다. 이러한 **임베딩 벡터들이 RAG의 검색 기능의 토대**가 됩니다.

- **벡터 저장소 (Vector Store)**: 임베딩한 벡터들을 저장하고 **유사도 검색**을 수행하는 **특수한 데이터베이스**입니다. 각 문서 청크의 임베딩 벡터를 키로 삼아 저장해두고, 나중에 쿼리 벡터가 들어오면 **벡터 공간에서 가장 가까운 벡터들을 빠르게 찾아**줍니다 [14]. 이를 일반 데이터베이스 대신 사용하는 이유는 벡터 간 코사인 유사도나 거리 기반으로 **텍스트 의미 유사성 검색**을 할 수 있기 때문입니다. 대표적인 벡터 DB로 FAISS, Chroma, Pinecone, Weaviate 등이 있으며, LangChain은 이들에 대한 **일관된 인터페이스로 VectorStore 클래스를 제공**하여 쉽게 벡터 저장소를 다룰 수 있습니다.

- **리트리버 (Retriever)**: **주어진 쿼리에 대해 적합한 문서 조각들을 찾아주는 검색 모듈**을 가리킵니다. 벡터 저장소가 데이터를 물리적으로 보관하는 역할이라면, 리트리버는 **해당 저장소를 질의하여 가장 관련도 높은 문서들의 목록을 반환**하는 역할을 합니다 [15]. 마치 질문에 답하기 위해 **관련 자료를 찾아오는 사서나 검색엔진**에 해당합니다. LangChain의 리트리버는 VectorStore 외에도 BM25 같은 전통 검색이나 Hybrid 검색 등 다양한 방식으로 구현될 수 있으며, **統一된 인터페이스 (`get_relevant_documents` 등)**를 통해 사용됩니다 [16]. 요약하면, 리트리버는 **쿼리를 입력으로 받아 관련 문서를 출력하는** 중간자입니다.

- **LLM (대형 언어 모델)**: GPT-4와 같은 **매우 거대한 규모의 언어 모델**로, **사람처럼 자연스러운 텍스트를 생성**할 수 있는 인공지능 모델입니다. 방대한 양의 텍스트로 사전 학습되어 **질문 답변, 번역, 글쓰기 등 다양한 언어 작업을 수행**할 수 있습니다. RAG 시스템에서 LLM은 최종적으로 **프롬프트(질문 + 컨텍스트)를 받아 답변을 생성하는 역할**을 담당합니다. 본래 LLM 단독으로는 학습 데이터 이후의 새로운 정보를 알지 못하지만, RAG에서는 앞 단계에서 **제공된 컨텍스트를 활용**함으로써 훨씬 **정확하고 구체적인 응답**을 만들 수 있게 됩니다 [7].

- **프롬프트 템플릿 (Prompt Template)**: **LLM에게 줄 프롬프트 양식을 사전에 정의해 놓은 템플릿**입니다. 매번 질문과 문맥을 문자열로 이어 붙이는 대신, 일정한 형식의 틀을 만들어두고 필요한 부분에 변수만 채워 넣는 방식입니다. 예를 들어 "다음 문맥을 참고하여 질문에 답하되, 모르면 '모르겠다'고 답하세요.\n문맥: {context}\n질문: {query}\n답변:" 와 같은 템플릿을 만들어 두고 `{context}`와 `{query}`만 채우는 식입니다 [17] [18]. 이렇게 하면 **프롬프트 구조의 일관성**을 유지하고 여러 가지 질문에 대해 **손쉽게 재사용**할 수 있습니다 [19]. LangChain의 `PromptTemplate` 클래스는 이러한 템플릿을 지원하며, 문자열 포맷에 placeholders를 넣어 **여러 입력 변수를 깔끔하게 주입**하는 기능, 템플릿 유효성 검사 등의 편의성을 제공합니다. 잘 설계된 프롬프트 템플릿은 **LLM의 출력 품질과 일관성**을 높이는 데 매우 중요합니다.

- **체인 (Chain)** 및 **파이프라인 (Pipeline)**: **여러 구성 요소를 순차적으로 엮어 하나의 작업 흐름으로 만든 구조**를 말합니다. LangChain에서 체인은 **프롬프트 생성 -> LLM 호출 -> 응답 처리**와 같은 일련의 단계를 **하나의 호출로 묶은 객체**로 볼 수 있습니다 [20]. 예를 들어, **RetrievalQA 체인**은 "질문을 받아 -> 리트리버로 관련 문서를 찾고 -> 해당 문서를 포함한 프롬프트를 생성하여 LLM에 보내 -> 최종 답변을 반환"까지의 과정을 한 번에 처리합니다. 체인을 사용하면 각 단계를 개별적으로 호출할 필요 없이 **마치 하나의 함수처럼 사용할 수 있어 코드가 간결해지고 모듈화**됩니다. 파이프라인이라는 용어도 비슷한 의미로 쓰이며, **데이터 흐름을 따라 여러 컴포넌트가 연속적으로 동작하는 구조**를 가리킵니다. LangChain에서는 체인/파이프라인을 유연하게 구성할 수 있어, 개발자가 필요한 단계들을 자유롭게 조합할 수 있습니다.

- **메모리 (Memory)**: **체인 실행 간에 상태를 저장하여 컨텍스트를 유지하는 기능**입니다. 특히 챗봇처럼 **대화형 응용**에서, 이전에 사용자가 한 말을 기억하고 대화를 이어가기 위해 메모리를 활용합니다. LangChain의 Memory 모듈은 **이전 대화 내역을 저장**해 두었다가 새로운 프롬프트에 과거 내용을 포함시키는 방식으로 동작합니다 [21]. 예를 들어 `ConversationBufferMemory`는 모든 대화 기록을, `ConversationSummaryMemory`는 요약된 기록을 관리합니다. RAG와 메모리를 함께 사용하면, **사용자 질의에 대한 관련 문맥 자료 + 이전 대화의 맥락**을 모두 고려하여 LLM이 답변하게 할 수 있습니다. 단, 메모리를 사용할 때는 대화 기록이 너무 길어지지 않도록 요약하거나 토큰 수를 제한하는 등 관리가 필요합니다.

- **쿼리 (Query)**: 사용자가 묻는 **질문이나 검색어**를 의미합니다. RAG 파이프라인은 이 쿼리를 입력으로 받아 동작을 시작합니다. 쿼리는 평문 문자열 형태로 들어오며, 내부적으로는 임베딩 벡터로 변환되어 벡터 스토어에서 **관련 문서를 찾는 키**로 활용됩니다. 다시 말해, **쿼리는 사용자의 정보 요구를 표현한 것**이고, 리트리버는 이 쿼리를 토대로 지식 베이스에서 **관련성이 높은 문서들을 검색**합니다 [22] [23]. 좋은 결과를 얻으려면 쿼리가 명확해야 하지만, LangChain에서는 `MultiQueryRetriever`처럼 **하나의 질문을 자동으로 변형하여 여러 쿼리로 검색**함으로써 사용자의 모호한 질문도 폭넓게 탐색하는 기법 등도 제공합니다.

- **맥락 압축 (Contextual Compression)**: 검색된 문서들 중 **질문에 직접 관련된 정보만 추려서 컨텍스트로 사용하는 기법**입니다. 일반적인 RAG에서는 리트리버가 찾아준 문서 조각들을 그대로 모두 프롬프트에 넣지만, 때로는 이들 중 일부 내용만 질문과 관련되고 나머지는 불필요할 수 있습니다. **Contextual Compression은 주어진 질의에 비추어 각 문서의 중요 부분만 남기거나 아예 중요하지 않은 문서는 제외**함으로써, **프롬프트에 담기는 컨텍스트를 압축**해 줍니다 [24]. 예를 들어 압축을 위해 **문서 요약 모델을 사용**하거나, **사전에 정의한 필터 규칙**으로 relevance가 낮은 문서를 거르는 방식이 있습니다 [25] [26]. 이 기능을 쓰면 LLM에 전달하는 토큰 수를 줄여 **비용을 절감하고 답변 정확도를 높일 수 있지만**, 압축 과정에서 정보 유실이 없도록 주의해야 합니다. LangChain은 `ContextualCompressionRetriever` 등을 통해 이러한 맥락 압축을 지원합니다.

LangChain을 활용한 RAG 개발의 간소화

LangChain 프레임워크는 앞서 소개한 RAG의 구성 요소들을 일일이 저수준에서 구현하지 않고도 쉽게 사용할 수 있게 해주는 도구 모음이라고 볼 수 있습니다. LangChain이 RAG 개발을 간소화하는 방법을 몇 가지로 정리하면 다음과 같습니다:

- **다양한 문서 로더 제공**: 파일 형식이나 데이터 소스별로 이미 구현된 Document Loader를 제공하므로, PDF, 워드, 웹 페이지, 데이터베이스 등 **여러 출처의 데이터를 간단한 메소드 호출로 불러올 수 있습니다** [8].

- **자동 청크 분할 도구**: `TextSplitter` 모듈을 통해 **문서 길이에 따른 청크 분해를 손쉽게 적용**할 수 있습니다. 토큰 단위, 문자 단위, 문단/문장 단위 등 **여러 분할 전략을 옵션으로 선택**할 수 있어, 적절한 크기로 문서를 나누는 작업이 용이합니다.
- **임베딩 및 벡터스토어 통합**: LangChain은 OpenAI, HuggingFace 등 **여러 임베딩 모델 래퍼**를 제공하고, FAISS, Chroma, Pinecone 등 **벡터 데이터베이스와 바로 연동되는 인터페이스**를 갖추고 있습니다. 예를 들어 `FAISS.from_documents(documents, embedding)` 한 줄로 문서들을 임베딩하여 FAISS 벡터스토어에 삽입하고 검색까지 가능하게 설정할 수 있습니다.
- **통합된 리트리버 인터페이스**: `vectorstore.as_retriever()` 같은 메소드로 **벡터스토어를 곧바로 리트리버로 사용할 수 있고**, BM25나 Hybrid 검색용 리트리버도 별도로 구현되어 있습니다. 따라서 복잡한 검색 로직을 직접 코딩하지 않고도 **간단한 함수 호출로 원하는 검색 방식을 적용**할 수 있습니다.
- **체인으로 구성 요소 결합**: LangChain에는 **미리 구성된 RAG 체인**이 있어, 몇 줄의 코드로 전체 RAG 파이프라인을 실행할 수 있습니다. 예를 들어 `RetrievalQA` 체인을 사용하면 리트리버와 LLM만 지정하여 **"질문 -> 검색 -> 답변"** 흐름을 바로 구현할 수 있습니다. 또한 대화형 봇을 만들 때는 `ConversationalRetrievalChain`을 사용해 **메모리까지 포함한 RAG 체인**을 쉽게 구축할 수 있습니다.
- **메모리 및 기타 부가 기능**: LangChain은 **대화 메모리 관리**, **출력 파서**, **토큰 사용량 모니터링**, **콜백 로깅** 등 실용적인 기능들을 내장하고 있어서, RAG 기반 애플리케이션을 **프로덕션 수준으로 개발하는 데 필요한 도구들을 한 곳에서 지원**합니다 [27] . 이러한 풍부한 기능 덕분에 개발자는 RAG의 개념적인 흐름에 집중할 수 있고, 세부 구현은 LangChain 라이브러리가 대신 처리해줍니다.

아래는 LangChain을 활용하여 **간단한 RAG 질의응답 시스템**을 구성하는 예시 코드입니다. 주석으로 각 단계를 설명하였으며, 이처럼 LangChain의 모듈들을 조합하여 최소한의 코드로 RAG 파이프라인을 구축할 수 있습니다:

1. **문서 로드 및 분할:** 예시로 간단한 텍스트 파일을 로드한 뒤, `CharacterTextSplitter`를 이용해 청크 단위로 분할합니다.

```
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter

loader = TextLoader("example.txt", encoding="utf8")
documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=500, chunk_overlap=50)
docs = text_splitter.split_documents(documents)
print(f"총 {len(docs)}개의 청크로 분할되었습니다.")
```

2. **벡터스토어 생성:** 분할된 문서 조각들에 대해 임베딩을 생성하고, FAISS 벡터스토어에 저장합니다. 그런 다음 벡터스토어로부터 리트리버 객체를 얻습니다.

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS

embeddings = OpenAIEmbeddings()  # OpenAI 임베딩 모델 (예시)
vector_store = FAISS.from_documents(docs, embedding=embeddings)
retriever = vector_store.as_retriever(search_kwargs={"k": 5})
```

3. **LLM 및 QA 체인 구성:** OpenAI의 GPT-3.5-turbo 모델을 LLM으로 사용하고, LangChain의 `RetrievalQA` 체인을 통해 질의응답 시스템을 생성합니다. 리트리버를 체인에 넘겨주면 질문에 대해 자동으로 벡터 DB를 검색하여 답변하게 됩니다.

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

llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
qa_chain = RetrievalQA.from_chain_type(llm, chain_type="stuff", retriever=retriever)
# chain_type="stuff": 검색 결과를 모두 프롬프트에 그대로 넣는 방식
```

4. **질의 실행:** 준비된 QA 체인에 사용자의 질문을 입력하여 답변을 얻습니다. 아래 예에서는 임의의 질문으로 지식베이스에서 답변을 생성합니다.

```
query = "예시 지식 베이스에 대한 질문을 여기에 넣으세요."
result = qa_chain.run(query)
print("답변:", result)
```

위 코드 예시는 LangChain의 모듈들을 활용하여 **문서 로드 → 분할 → 임베딩 및 저장 → 검색 → 답변 생성**의 RAG 파이프라인을 구현한 것입니다. 불과 몇 줄의 코드로 원하는 데이터를 기반으로 하는 질의응답 시스템을 구축할 수 있으며, 필요에 따라 체인의 구성을 변경하거나 메모리를 추가하는 등 확장이 가능합니다.

마치며

지금까지 **RAG의 개념과 중요성**, 그리고 **LangChain을 활용한 RAG 구성 요소와 개발 방법**을 살펴보았습니다. 핵심 요점을 정리하면 다음과 같습니다:

- **RAG**는 LLM의 한계를 극복하기 위해 외부 지식을 검색하여 답변에 활용하는 방법으로, **정확하고 최신의 응답**을 얻는 데 유용합니다 [4].
- RAG 시스템은 **문서 로드 → 청크 분할 → 임베딩 & 벡터스토어 저장 → 쿼리 임베딩 → 유사도 검색 → 프롬프트 구성 → LLM 생성**의 **파이프라인**으로 동작하며, 각 단계에 **Document Loader, Text Splitter, Embeddings, Vector Store, Retriever, Prompt Template, LLM** 등의 컴포넌트가 사용됩니다.
- **LangChain** 라이브러리는 이러한 RAG 구현을 크게 단순화해주며, 이미 준비된 **통합 인터페이스와 모듈**들을 통해 **초보자도 손쉽게 RAG 기반 앱을 개발**할 수 있게 해줍니다 [27].
- 궁극적으로 RAG를 활용하면 **신뢰할 수 있는 출처에 기반한 응답**을 생성함으로써 사용자에게 **투명하고 신뢰도 높은 AI 응용 서비스**를 제공할 수 있습니다. RAG와 LangChain을 활용한 여러분만의 프로젝트에 도전해 보세요!

참고 자료: RAG와 LangChain의 구성 요소에 대한 공식 문서와 블로그 글을 함께 참고하면 이해를 더욱 높일 수 있습니다 [1] [2] [3] [4] [15] [9] 등. 이 문서에 포함된 인용 표시는 해당 내용을 출처에서 발췌했음을 나타냅니다. 보다 깊이 있는 내용은 각 출처를 직접 확인해보시기 바랍니다.

---

[1] [2] [3] [6] What is RAG? - Retrieval-Augmented Generation AI Explained - AWS
https://aws.amazon.com/what-is/retrieval-augmented-generation/

[4] [5] [7] Retrieval-Augmented Generation (RAG) | Pinecone
https://www.pinecone.io/learn/retrieval-augmented-generation/

[8] [27] What are Langchain Document Loaders? - Analytics Vidhya
https://www.analyticsvidhya.com/blog/2024/07/langchain-document-loaders/

[9] [10] [11] Text splitters | LangChain
https://python.langchain.com/docs/concepts/text_splitters/

12 13 14 What Are Embeddings? How They Help in RAG - DEV Community
https://dev.to/shaheryaryousaf/what-are-embeddings-how-they-help-in-rag-2l1k

15 22 23 Understanding the Key Components of RAG: Retriever and Generator - DEV Community
https://dev.to/shaheryaryousaf/understanding-the-key-components-of-rag-retriever-and-generator-1a1j

16 RAG using LangChain : Part 4-Retrievers | by Jayant Pal | Medium
https://jayant017.medium.com/rag-using-langchain-part-4-retrievers-79908145e28c

17 18 19 Getting Started with LangChain Prompt Templates | Codecademy
https://www.codecademy.com/article/getting-started-with-lang-chain-prompt-templates

20 A Complete Guide of Chain with LangChain | by Mdabdullahalhasib | Medium
https://medium.com/@abdullah.iu.cse/a-complete-guide-of-chain-with-langchain-74e359b936d7

21 memory — LangChain documentation
https://python.langchain.com/api_reference/langchain/memory.html

24 25 26 How to do retrieval with contextual compression | LangChain
https://python.langchain.com/docs/how_to/contextual_compression/