ChatGPT

LLM을 활용한 AI 에이전트 개발 입문: 혼자 공부 가이드

이 가이드는 *"Do it! LLM을 활용한 AI 에이전트 개발 입문"* 책의 내용을 토대로, 혼자서 단계별로 실습하고 학습할 수 있도록 정리한 **주피터 노트북 스타일 자료** 입니다. Python과 주피터 노트북 환경에 익숙하지만, 컴퓨터 공학 전공자는 아니며 LangChain 라이브러리를 처음 접하는 독자를 염두에 두었습니다. VS Code의 Jupyter 확장과 OpenAI 호환 API를 사용할 수 있는 환경을 가정하며, **각 섹션마다 개념 설명과 함께 핵심 코드 예시**를 포함하였습니다. 회사 등 보안 제한이 있는 환경에서 실습할 경우를 대비하여 **특수 상황에서의 대처법**도 함께 설명합니다.

이 가이드를 따라가다 보면 **GPT API 활용 기본기부터, 텍스트/음성/이미지 처리, LangChain을 이용한 대화 메모리와 툴 통합, 검색 기능 연계, 최종적으로는 여러 에이전트를 조합한 고급 프로젝트 구현**까지 한걸음씩 배우게 됩니다.

OpenAI API 기본 사용법

시작하기에 앞서, OpenAI의 GPT API(챗GPT API)를 호출하는 기본 방법을 알아봅시다. OpenAI 계정에서 API 키를 발급받아 환경변수 `OPENAI_API_KEY`에 설정하거나 코드에서 직접 설정해야 합니다. 주피터 노트북에서는 다음과 같이 키를 지정할 수 있습니다:

```
import openai
openai.api_key = "발급받은-나의-API키"
```

또는 `.env` 파일이나 OS 환경변수를 사용하는 방법도 있습니다 (예: `python-dotenv` 사용). 회사 내 폐쇄망 등에서 자체 호환 API를 쓰는 경우, OpenAI 라이브러리의 `api_base` 엔드포인트를 변경해주는 방식으로 설정할 수 있습니다. 예를 들어 내부 호환 API URL이 `https://corp-api.example.com/v1`라면 `openai.api_base = "https://corp-api.example.com/v1"` 과 같이 지정하면 호환됩니다.

모델 선택: OpenAI API에서는 `"gpt-3.5-turbo"`, `"gpt-4"` 등의 모델명을 사용합니다. 책의 코드에서는 `gpt-4` 기반 사설 모델 명시를 위해 `"gpt-4o"`처럼 접미사를 붙여 표기했으나(예:), 실제로는 **사용 권한이 있는 모델 명**을 입력해야 합니다. 예제를 실행할 때 접근 가능한 모델로 변경하세요.

한글 입출력: GPT API는 기본적으로 다국어를 지원하므로 프롬프트와 응답을 한글로 주고받을 수 있습니다. 다만 정확한 결과를 얻기 위해 **역할(Role) 지시어와 구체적인 지침**을 프롬프트에 명시하는 것이 중요합니다. 예를 들어 시스템 프롬프트에 AI의 역할과 출력 형식을 한국어로 상세히 알려주는 식입니다.

이제 본격적인 각 챕터별 주제 학습을 시작하겠습니다.

문서 요약: PDF 텍스트 처리와 LLM 요약

대용량 텍스트 요약은 LLM 활용의 대표 사례입니다. 책의 4장에서는 **연구 보고서 PDF 문서**를 읽어 요약하는 과정을 다룹니다. 주요 단계는 **PDF -> 텍스트 변환**과 **텍스트 -> 요약 생성**으로 나뉩니다.

- **PDF를 텍스트로 추출:** `PyPDF2`나 `pdfplumber` 등의 라이브러리를 이용해 PDF 파일의 본문을 추출합니다. 책의 예제 코드는 PDF 페이지를 순회하며 텍스트를 모은 뒤 전처리(헤더/푸터 제거 등)를 수행했습니다. 추출한 원문 텍스트는 용량이 크므로, 필요시 **요약에 불필요한 부분(페이지 번호, 각주 등)을 제거**합니다.

- **LLM에게 요약 요청:** 미리 준비한 시스템 프롬프트에 전체 텍스트를 포함시켜 GPT에게 요약을 부탁합니다. 아래는 책 예제의 요약 프롬프트 구성 방식입니다:

```
# 전체 텍스트 파일을 읽어 문자열로 불러옴
with open(file_path, 'r', encoding='utf-8') as f:
    txt = f.read()

system_prompt = f'''
너는 다음 글을 요약하는 봇이다. 아래 글을 읽고, 저자의 문제 인식과 주장을 파악하고, 주요 내용을 요약하라.

작성해야 하는 포맷은 다음과 같다.
# 제목
## 저자의 문제 인식 및 주장 (15문장 이내)
## 저자 소개

============== 이하 텍스트 ==============
{txt}
'''
```

이 프롬프트에서 요약 **출력 형식**을 제목과 소제목 형태로 요구하고, 요약해야 할 본문 내용을 `==========` 아래에 첨부하고 있습니다. 이후 GPT에게 챗 완료(create completion) 호출을 하면 요약 결과를 받을 수 있습니다.

※ **참고:** 이렇게 텍스트 전체를 프롬프트로 보내는 방식은 토큰 한도 내에서만 가능합니다. GPT-4 기준 수천 단어까지는 가능하나, 더 큰 문서는 잘라서 요약 후 모으는 방식 등이 필요합니다.

- **응답 처리:** GPT가 생성한 요약 결과를 파일로 저장하거나 출력합니다. 책의 예시에서는 요약 결과를 `crop_model_summary.txt`로 저장했습니다.

예를 들어, 위 과정으로 **농업 작물모형 보고서** PDF를 요약하면, GPT가 보고서의 핵심 내용을 뽑아 한글 요약문을 만들어줍니다. 이때 프롬프트에서 지정한 형식대로 제목과 섹션을 포함해 답변하도록 유도한 것이 포인트입니다.

실습 팁: 데이터 파일 경로와 인코딩

PDF에서 추출한 텍스트를 요약할 때 **파일 경로와 인코딩에 주의** 해야 합니다. 경로에 한글이 포함된 경우 파이썬에서 `open()`으로 읽을 때 Unicode 경로 처리가 필요할 수 있습니다. 또한 PDF 추출 라이브러리에 따라 결과 텍스트의 인코딩이 UTF-8이 아닐 수 있으므로, `read()`할 때 `encoding='utf-8'`을 명시하는 것이 좋습니다.

회사 내부 환경 특이사항

회사 환경에서 인터넷이 차단된 경우라도, **PDF 파일을 로컬로 확보**하면 요약 작업을 수행할 수 있습니다. 다만, 책의 예제 데이터 중 9장 RAG 실습에서는 두 개의 PDF가 필요하지만 저작권 문제로 저장소에 빠져 있습니다. 이 파일들은 공개 웹에서 내려받아야 하는데, 사내 PC에서 해당 사이트 접속이 막힌 경우 **다음과 같이 우회**할 수 있습니다:

- **외부에서 파일을 받아오는 방법:** 회사 밖 인터넷이 되는 곳에서 PDF를 다운로드하여 USB 등으로 가져오거나, 사내망에서 `requests` 등을 이용해 파일을 내려받습니다 (방화벽이 허용한다면). 예를 들어:

```
import requests
url = "https://example.com/OneNYC_2050_Strategic_Plan.pdf"
r = requests.get(url)
with open("OneNYC_2050_Strategic_Plan.pdf", "wb") as f:
    f.write(r.content)
```

- **대체 데이터로 실습:** 만약 해당 보고서를 구하기 어려운 경우, **비슷한 텍스트 분량의 다른 공개 문서**로 대체해서 실습해도 무방합니다. 중요한 것은 PDF를 처리해 벡터DB를 만드는 과정이지 특정 문서 자체는 아니므로, 예를 들어 공개 위키백과 문서를 여러 개 PDF로 저장해 사용하는 방법도 있습니다.

이렇게 PDF 요약을 통해 LLM 사용법을 익혔다면, 다음은 멀티모달 입력의 첫걸음으로 **음성 데이터를 처리**하는 방법을 알아보겠습니다.

음성 인식과 요약: Whisper를 활용한 STT

LLM 기반 에이전트는 텍스트 뿐 아니라 **음성 데이터를 이해**하여 활용할 수도 있습니다. 5장에서는 음성 녹음 파일을 받아 **1) 텍스트로 변환(STT)**하고, **2) 중요한 내용 요약**까지 진행하는 예제를 다룹니다. OpenAI의 Whisper 모델과 Hugging Face의 Whisper 구현을 모두 활용합니다.

- **OpenAI Whisper API 사용:** OpenAI에서 제공하는 Whisper는 고품질 음성-텍스트 변환 모델입니다. OpenAI API 키로 Whisper 엔드포인트를 호출하면 비교적 간단히 STT를 수행할 수 있습니다. 책의 예제 `openai_api_whisper.ipynb`에서는 `openai.Audio.transcribe("whisper-1", audio_file)` 형태로 사용하여, MP3 파일을 업로드하고 한글 음성을 텍스트로 변환했습니다.

```
import openai
audio_file = open("meeting_record.mp3", "rb")
transcript = openai.Audio.transcribe("whisper-1", audio_file)
print(transcript['text'])
```

OpenAI Whisper의 장점은 **별도 모델 다운로드 없이** 곧바로 높은 정확도의 결과를 얻을 수 있다는 것이지만, 사내망에서는 이 방법이 불가능할 수 있습니다. 또한 API 호출 비용과 인터넷 업로드가 발생합니다.

- **Hugging Face Whisper 사용:** 인터넷 연결이나 외부 API 사용이 어려운 경우, 사내에서 직접 오픈소스 Whisper 모델을 돌릴 수 있습니다. 책의 `huggingface_whisper.ipynb` 예제는 `transformers`와 `datasets` 라이브러리를 활용하여 미리 학습된 Whisper 모델(예: `openai/whisper-small`)을 로컬에서 실행했습니다. 이 방식은 처음에 모델을 다운로드해야 하지만, 이후로는 오프라인으로도 동작합니다. 주피터 노트북 환경에서:

```
from transformers import pipeline
stt_pipeline = pipeline(task="automatic-speech-recognition",
                        model="openai/whisper-small", device=0)
result = stt_pipeline("meeting_record.mp3")
text = result['text']
```

위와 같이 파이프라인을 구성해 `meeting_record.mp3` 파일을 변환하면 텍스트를 얻을 수 있습니다. **주의:** 이때 `device=0`는 GPU 사용을 지정한 것으로, GPU가 없는 환경에서는 제거하거나 `device=-1`로 CPU 모드로 전환하세요. Whisper는 **연산량이 많아** CPU로 처리하면 느릴 수 있습니다.

- **발화자 분리 (화자 다중화, Diarization):** 책에서는 대화 녹음에서 **여러 명의 목소리를 구분**하는 심화 주제로 **PyAnnote.audio** 라이브러리를 활용합니다. PyAnnote의 사전학습 모델 `pyannote/speaker-diarization`을 통해, 음성 파일에서 음성이 나온 구간과 화자를 구분하는 기능을 구현했습니다. 이 부분은 `speaker_diarization.ipynb`에 포함되어 있으며, `Pipeline.from_pretrained("pyannote/speaker-diarization-3.1")`으로 모델을 불러와 사용합니다.

이때 Hugging Face로부터 모델을 처음 다운로드하는데, 회사 환경에서 해당 요청이 차단된다면 **사전에 모델 파일을 받아와 로컬에 캐시**해야 합니다. (예: 개발 PC 등에서 `pyannote.audio` 모델을 한번 다운로드한 뒤, 그 캐시 디렉토리를 오프라인 환경에 복사.) PyAnnote 모델은 용량이 큰 편이므로 네트워크 제한을 특히 유념해야 합니다.

- **문서 요약/정리:** 음성을 텍스트로 변환한 후 얻은 대화록에 대해, 이전 섹션처럼 GPT를 이용해 요약하거나 **중요 내용 추출**을 수행할 수 있습니다. 책 예제에서는 회사 회의록을 Whisper로 받아 적은 뒤, GPT에게 해당 회의의 주요 논의와 결정 사항을 요약하도록 했습니다. 이렇게 **STT + 요약** 파이프라인을 구성하면, 음성 회의록을 자동으로 정리하는 AI 비서 기능을 만들 수 있습니다.

실습 팁: 긴 오디오의 처리

긴 음성 파일은 수 분 단위로 잘라서 Whisper에 넣거나, Whisper 모델이 **긴 컨텍스트를 허용하는 버전**을 사용해야 합니다. Hugging Face의 `openai/whisper-large-v2` 등은 약 30초~1분 단위 처리에 적합합니다. 만약 더 긴 녹음을 한꺼번에 처리하려면, **segment** 단위로 자른 뒤 텍스트를 이어붙여야 합니다. 책의 예제 코드에서는 Whisper로 변환된 텍스트를 바로 요약했지만, 필요하다면 **화자 구분 결과(RTTM 파일)**를 활용해 `"사람 A:"`, `"사람 B:"` 식으로 대화 형태로 포맷팅한 후 GPT에게 요약을 요청하는 응용도 가능합니다.

회사 내부 환경 특이사항

- **Hugging Face 모델 다운로드 차단:** Whisper나 PyAnnote 모델을 Hugging Face에서 내려받아야 하는 경우, 앞서 언급한 대로 **모델 파일을 미리 수동으로 준비** 해야 합니다. 예컨대 Whisper-small 모델은 수백 MB 규모의 `.bin` 파일이며, PyAnnote diarization 모델도 수백 MB 이상입니다. 미리 인터넷 연결 되는 환경에서

`pipeline(..., model="모델이름")`을 한 번 실행하면 `~/.cache/huggingface` 등에 캐시가 저장되니, 그 폴더를 복사하는 방법도 있습니다.

- **대안 API 사용:** 만약 사내에 자체 STT 엔진이 있거나, Google Speech-to-Text 같은 대체 수단이 허용된다면, 해당 API를 호출하는 방식으로 녹음 파일을 텍스트로 변환하고 이후 과정은 동일하게 GPT 요약을 적용할 수도 있습니다.

다음으로는 **이미지 데이터를 LLM과 함께 활용**하는 방법을 알아보겠습니다.

이미지 분석과 활용: 시각 정보 처리

6장에서는 이미지 파일을 다루며, AI 에이전트가 **이미지 내용을 설명**하거나 **이미지를 보고 질문에 답하는** 기능을 구현합니다. 최근 GPT-4에 이미지 입력 기능이 등장했지만, 이 책에서는 오픈소스 도구와 결합하여 이미지 처리를 수행했습니다.

- **이미지 캡션 생성:** 먼저 **이미지 내용을 텍스트로 묘사**하는 모델이 필요합니다. Hugging Face의 *BLIP2*, *CLIP* 모델 등이 대표적입니다. 예제 `image_explanation.ipynb`에서는 미리 학습된 Vision-Encoder와 GPT 계열 Decoder를 결합한 BLIP2 모델을 사용하여 입력 이미지(`.jpg`)에 대한 설명문을 얻었습니다. 사용 방식은 다음과 비슷합니다 (Pseudo-code):

```
from transformers import BlipProcessor, BlipForConditionalGeneration
processor = BlipProcessor.from_pretrained("Salesforce/blip2-flan-t5-xl")
model = BlipForConditionalGeneration.from_pretrained("Salesforce/blip2-flan-t5-xl")
image = Image.open('busan_dive.jpg')
inputs = processor(image, "사진 설명를 한국어로 자세히 써줘.", return_tensors="pt")
result = model.generate(**inputs)
description = processor.decode(result[0], skip_special_tokens=True)
print(description)
```

위와 같은 과정을 거치면, 예를 들어 부산 다이버 커피숍 사진(`busan_dive.jpg`)을 입력했을 때 "해변가 작은 카페에서 사람들이 커피를 마시고 있다..." 와 같이 장면을 묘사한 문장을 얻을 수 있습니다. **주의:** 실제 코드에서는 모델명과 프롬프트를 정확히 지정해야 하고, 한글 출력을 위해 프롬프트를 한국어로 주거나 결과를 번역하는 단계를 추가할 수 있습니다.

- **LLM과 결합:** 이미지 캡션만으로 끝나는 것이 아니라, LLM에게 **이미지에 대한 추가 설명이나 질의 응답**을 시킬 수 있습니다. 예제 `image_quiz.py`에서는 이미지 설명을 이용해 퀴즈를 내는 에이전트를 구현했습니다. 흐름은 이렇습니다:
- **이미지로부터 1차 캡션**을 얻는다 (예: "이미지에는 해변 카페와 두 사람이 있다").
- **추가 프롬프트 구성:** GPT의 시스템 메시지로 "너는 사진 설명 전문가다. 주어진 사진 설명을 바탕으로 사용자에게 재미있는 질문을 해라." 등을 넣고, 사용자 역할 메시지에 1차 캡션을 첨부합니다.
- **GPT 응답 생성:** GPT는 사진에 대한 퀴즈 형태 질문이나 창의적 설명을 생성합니다.

이러한 식으로, **이미지 -> 텍스트 설명 -> LLM 활용**의 멀티모달 파이프라인을 구축할 수 있습니다.

- **OCR 활용:** 이미지 내에 **텍스트가 포함된 경우** (예: 스캔 문서, 사진 속 간판 등), Vision-to-Text 대신 **OCR(광학문자 인식)** 기술을 적용해야 합니다. 책에서 직접 다루지는 않았지만, 필요하다면 `pytesseract` 같은 OCR 도구로 이

미지를 처리한 뒤 나온 텍스트를 LLM이 분석하도록 할 수 있습니다. 예를 들어 사내 문서 스캔 이미지라면 OCR로 텍스트를 추출한 후 GPT에게 요약이나 분류를 요청하는 방식입니다.

실습 팁: 모델 경량화와 성능

BLIP2와 같은 이미지 캡션 모델은 **용량이 매우 크고** 실행에 시간이 많이 걸릴 수 있습니다. CPU로 실행하면 한 장 처리에도 수십 초 이상 걸릴 수 있으므로, 가능하다면 **GPU가 있는 환경**에서 하거나, **보다 경량 모델**(예: `Salesforce/blip2-flan-t5-base` 등)을 선택하는 것이 좋습니다.

또한 한번 캡션을 생성한 이미지는 **결과를 캐싱**해두면 반복 사용시 시간을 절약할 수 있습니다. 예를 들어 동일 이미지에 대해 여러 번 질문을 할 경우 매번 Vision 모델을 돌리지 않고 처음 캡션만 저장해두고 활용하면 효율적입니다.

회사 내부 환경 특이사항

- **모델 다운로드:** 이미지 캡션을 위한 모델 역시 Hugging Face에서 받아와야 하므로, 앞서 Whisper 경우처럼 미리 모델 파일을 준비해야 합니다. 용량이 매우 크기 때문에(수 GB 이상) 사전에 네트워크 인프라팀과 논의가 필요할 수 있습니다.

- **대안 접근:** 만약 사내에 **이미지 인식 API**가 있다면, 예를 들어 사내 비전 모델이 반환하는 태그 목록이나 캡션을 활용하는 것도 방법입니다. 완벽한 문장형 캡션이 아니어도, LLM에게는 핵심 키워드만 있어도 충분히 묘사를 풍부하게 만들 수 있기 때문입니다. 예를 들어 "사무실, 회의실, 사람 3명, 화이트보드" 같은 태그를 LLM에 주고 **이 정보를 바탕으로 장면을 추측**하게 프롬프트를 구성할 수도 있습니다.

지금까지 텍스트, 음성, 이미지 데이터를 **입력으로 처리하여 LLM과 결합**하는 방법을 살펴보았습니다. 다음은, LLM이 **외부 도구나 함수를 호출**하여 능력을 확장하는 방법을 알아보겠습니다.

GPT 함수 호출과 도구 통합

7장에서는 OpenAI가 제공하는 **Function Calling** 기능과, 이를 활용해 GPT가 **외부 함수를 호출하는 에이전트**를 만드는 법을 다룹니다. 이를 통해 LLM이 스스로 부족한 부분을 코드를 통해 보완하거나, 실시간 정보를 얻어오는 등 **툴 사용**이 가능해집니다.

- **Function Calling 개념:** GPT 모델에 함수의 사양(spec)을 미리 알려주면, 답변이 필요할 때 그 함수를 호출하는 형태로 응답을 생성할 수 있습니다. 예를 들어 "현재 시간을 알려주는 함수" 를 정의해두면 GPT는 사용자 질문에 곧장 시간을 알려주는 대신 함수 호출 포맷으로 응답하고, 클라이언트 측 코드가 그 함수를 실행한 결과(현재 시간)를 다시 GPT에게 주면 최종 답을 완성하는 식입니다. 이 절차를 통해 GPT가 **툴 사용 추론**을 하게 됩니다.

- **현재 시간 예제:** 책의 7.1절에서는 간단히 현재 시계를 알려주는 에이전트를 터미널과 Streamlit UI로 구현했습니다. 함수 정의는 예컨대 파이썬으로:

```
import datetime
def get_current_time():
    return datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

GPT에게는 이 함수의 이름과 목적, 리턴 형식을 JSON Schema로 설명해 줍니다. OpenAI `openai.ChatCompletion.create` 호출 시 `functions=[ ... ]` 파라미터에 해당 정의를 넣으면, GPT는 사용자의 "지금 몇 시야?" 같은 질문에 `{"function_call": {"name": "get_current_time", "arguments": ""}}` 형태의 답을 만들 수 있습니다. 클라이언트는 이를 파싱해 `get_current_time()` 함수를 실제로 실행하고, 결과 시간을 다시 메시지로 GPT에게 주면 비로소 `"현재 시간은 2025-07-26 01:17:02입니다."` 같은 최종 답변을 얻습니다.

- **주식 정보 예제:** 7.2절에서는 좀 더 실용적인 함수 호출 예로 **주식 시세 조회**를 다룹니다. `yfinance` 라이브러리를 통해 특정 주식의 현재 가격이나 정보를 가져오는 함수를 만들고, GPT에게 그 함수를 부를 수 있음을 알려줍니다. 사용자가 "애플의 오늘 주가가 어떻게 돼?" 물으면 GPT는 함수 호출 형태로 응답하고, 클라이언트가 `yfinance`로 데이터를 받아와 결과를 GPT에게 주면 GPT가 자연어로 설명하는 식입니다. 이렇게 하면 GPT가 모르는 실시간 정보도 툴로 획득해 답변에 활용할 수 있습니다.

- **스트리밍 응답:** 7.3절에서는 OpenAI API의 **스트리밍 모드**를 사용하여 함수 호출 결과도 실시간으로 표현하는 방법을 소개합니다. 예를 들어 주식 시세를 조회하는 동안에 진행 상황을 계속 출력하거나, 시간 지연이 있는 작업의 경우 GPT가 대화 중간중간 진행율을 보고하도록 구현할 수 있습니다. 스트리밍은 `openai.ChatCompletion.create(stream=True, ...)`로 가능하며, 책의 코드는 이를 활용해 터미널에서 토큰이 실시간 출력되는 모습을 구현했습니다.

실습 팁: 함수 호출 파라미터 설계

OpenAI 함수 호출을 사용할 때는 함수의 `name`과 `parameters` (JSON 스키마) 설계를 신중히 해야 합니다. **함수명을 GPT 답변에 직접 노출하지 않는 것이 좋고**, 인자 스키마에는 필요한 최소한의 정보만 포함합니다. 예를 들어 주식조회 함수에 `ticker`와 `date` 두 인자가 있다면, 둘 다 `type: string`으로 정의하고 어떤 형식인지 `description`으로 명시합니다. GPT는 이 description을 참고하여 정확한 인자를 채우려 시도합니다.

또한 **사용자 질문을 함수 호출로 처리할지 여부**를 결정하는 역할은 GPT 모델에 달려있으므로, 시스템 프롬프트에서 `"주식 관련 질문에는 주식조회 함수를 사용하고, 나머지에는 평범하게 답하라"` 같은 지침을 줄 수 있습니다. 책의 `stock_info_streamlit.py` 등에서는 이러한 흐름 제어를 코드로 구현했습니다.

회사 내부 환경 특이사항

- **HTTP 요청 등 네트워크 함수:** GPT에게 함수 권한을 주는 것은 곧 임의 코드 실행을 허용하는 것이므로, 보안에 주의해야 합니다. 특히 회사망 내 데이터를 조회하거나 사내 API를 호출하는 함수를 연결할 경우, GPT 프롬프트 유출 등에 각별히 신경써야 합니다. 예를 들어, GPT가 의도치 않은 함수를 호출하지 않도록 **function 목록을 엄격히 통제**하고, 함수 실행 단계에서 입력값을 검증하는 로직이 필요합니다.

- **사내 데이터베이스 연동:** 함수 호출 아이디어를 확장하면, GPT가 사내 DB 질의 함수를 호출하게 할 수도 있습니다. 이 경우 함수는 예를 들어 SQL 쿼리를 인자로 받아 실행하는 식이 될 텐데, 이는 SQL 인젝션 등 위험을 수반하므로 추천되지 않습니다. 대신 **DB 조회용 중간 API**를 만들어 GPT는 안전한 질의 파라미터만 전달하게 하는 방식이 바람직합니다.

다음으로, 이러한 개별 기능들을 좀 더 편리하게 구성할 수 있는 LangChain 프레임워크를 살펴보겠습니다.

LangChain 기초: LLMChain, 메모리, 프롬프트 템플릿

8장부터는 **LangChain** 라이브러리를 활용하여, LLM 기반 애플리케이션을 구조화하고 복잡한 대화 상태를 관리하는 방법을 배웁니다. LangChain은 파이썬에서 LLM을 쉽게 쓸 수 있도록 하는 고급 프레임워크로, **체인(Chain)** 개념과 다양한 모듈을 제공합니다.

- **LLMChain과 프롬프트 템플릿:** LangChain의 기본 단위인 LLMChain은 프롬프트 템플릿 + *LLM* 조합으로 이루어집니다. 예를 들어 `PromptTemplate("너는 친절한 답변 봇이다. 질문: {문의}\n답변:")`와 `OpenAI(model="gpt-3.5-turbo")`를 결합해 `LLMChain`을 만들면, `.run({"문의": "오늘 날씨 어때?"})` 호출만으로 완성된 답변을 얻을 수 있습니다. 책의 `langchain_chatbot.ipynb` 예제에서는 사용자의 입력을 받아 간단한 답변을 돌려주는 체인을 구현했습니다.

- **대화 메모리 (ConversationMemory):** LangChain을 쓰는 큰 이점 중 하나는 대화 기록을 관리하는 **메모리**를 쉽게 붙일 수 있다는 점입니다. 예를 들어 `ConversationBufferMemory`를 LLMChain에 연결하면, 체인이 실행될 때마다 **과거 대화 내역**을 프롬프트에 자동으로 포함시켜 줍니다. 책의 `langchain_message_history.ipynb`에서는 사용자의 여러 차례 질문에 AI가 앞 맥락을 기억하며 답하도록 구현했습니다.

```
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferWindowMemory
conversation = ConversationChain(
    llm=OpenAI(temperature=0),
    memory=ConversationBufferWindowMemory(k=3),  # 최근 3턴 기억
)
result = conversation.predict(input="안녕?")
# -> "안녕하세요! 무엇을 도와드릴까요?"
result = conversation.predict(input="내가 방금 뭐라고 했지?")
# -> AI는 대화 메모리를 참고해 사용자의 이전 말을 알려줌
```

이처럼 `ConversationChain`을 사용하면 복잡한 프롬프트 작성 없이도 지난 대화문맥이 자동 유지됩니다. Memory의 종류에 따라 전체 기억, 일부 창 기억, 요약본 기억 등 다양한 전략을 적용할 수 있습니다.

- **복합 프롬프트와 출력 파싱:** LangChain의 PromptTemplate은 Jinja2 스타일로도 쓸 수 있어, 시스템/사용자 역할 메시지 여러 개를 조합하거나 함수 호출 포맷 등을 생성하는 데 도움을 줍니다. 또한 출력 결과를 특정 형식으로 받아야 할 때 **Output Parser**를 정의할 수 있습니다. 책의 8.3절에서 다루는 Pydantic을 이용한 출력 파싱은, GPT 응답을 미리 정해둔 Pydantic 모델에 맞게 검증하고 객체로 변환하는 기법입니다. 이를 통해 LLM의 자유형 텍스트를 구조화된 데이터로 다룰 수 있습니다.

예를 들어, AI가 `{'answer': "서울", 'reason': "수도이므로"}` 형태의 JSON을 반환하게 프롬프트를 구성하고, OutputParser로 이 JSON을 파싱하여 파이썬 딕셔너리로 얻는 식입니다. LangChain의 `StructuredOutputParser`나 PydanticOutputParser 등을 활용하면, **LLM 출력의 신뢰성을 조금 더 높이는 효과**를 얻습니다.

실습 팁: LangChain 버전 호환

LangChain은 버전업이 매우 빈번한 라이브러리입니다. 책의 코드는 `langchain_core` 0.3대 버전 등을 사용하지만, 현재(2025년 기준) `langchain` 통합 패키지 0.4 이상으로 올라가 인터페이스가 다소 변경되었을 수 있습니다. 예컨대 `ConversationChain` 대신 `LLMChain` + `memory` 속성으로 동일 기능을 구현하거나, OutputParser 부분도 최신 문서를 참조해야 할 수 있습니다.

실습 시 **pip으로 책과 동일 버전을 설치** 하거나, 예제 코드에 맞춰 import 경로를 수정하는 것이 필요합니다 (예: `from langchain import OpenAI` 등).

회사 내부 환경 특이사항

- **API 키 관리:** LangChain 사용시에도 결국 OpenAI 등의 API 키가 필요하므로, 소스 코드에 키를 하드코딩하기보다는 `.env` 파일이나 환경변수를 사용하는 것이 좋습니다. 책 예제들은 `load_dotenv()`로 키를 불러오는 형태를 사용하고 있으니, 사내 환경에서 환경변수 세팅만 잘 되어 있으면 코드 수정 없이도 실행이 될 것입니다.

- **프롬프트 내 민감정보**: LangChain은 편리하지만, 프롬프트에 어떤 정보가 들어가는지 신경써야 합니다. 특히 사내 대화 내용을 메모리에 저장하거나 외부 LLM API로 보내는 경우, **중요한 내부 정보가 포함되지 않는지** 검토해야 합니다. 필요하다면 프롬프트 구성 시에 특정 키워드 또는 개인정보를 마스킹하거나, 메모리를 Custom하게 구현하여 일부만 전달하는 등 조치가 필요합니다.

다음 장에서는 LangChain을 활용하여 **툴 사용 에이전트**와 **검색 결합** 등을 구현해보겠습니다.

LangChain 도구와 에이전트: 검색 통합

8장의 후반부와 10장에서는 LangChain 에이전트를 이용해 **웹검색, 유튜브 요약 등 외부 정보와 LLM을 결합** 하는 방법을 보여줍니다.

- **LangChain Tool과 Agent:** LangChain에서는 외부 기능을 추상화한 **Tool** 클래스와, 그 Tool을 언제 어떻게 쓸지 판단하는 **Agent**를 제공합니다. 예를 들어 `DuckDuckGoSearchAPI`를 툴로 등록하면, 에이전트는 사용자 질문에 답하기 위해 필요시 검색 툴을 호출하고 결과를 받아 다시 LLM에게 넘겨 답변에 활용할 수 있습니다.

책의 10.1절 `duckduckgo_search.ipynb` 예제는 DuckDuckGo 검색 툴을 사용해 **실시간 웹 정보에 답하는** 에이전트를 구현했습니다. LangChain의 `initialize_agent` 함수를 사용하여, LLM과 사용 가능한 툴 목록(`tools=[search]`)을 지정하고 `agent="zero-shot-react-description"` 모드를 선택하면, GPT가 **ReAct 스타일**로 생각하며 필요시 툴을 쓸 수 있게 됩니다. - 예: 사용자 질문: "월드컵 우승국 알려줘". 에이전트는 내부적으로 “검색이 필요하다” 판단 -> DuckDuckGoTool 실행 -> 결과 중 우승국을 찾아 최종 답 생성.

- **검색 결과 활용:** 검색 툴 사용의 핵심은 **LLM이 직접 웹 컨텐츠를 읽을 수 있게 하는 것** 입니다. DuckDuckGo 검색 툴의 결과는 보통 웹 페이지 링크와 snippet이지만, LangChain 에이전트는 한 단계 더 나아가 **웹페이지 내용도 가져와 읽게** 할 수 있습니다. 예제에서는 `serpapi` 등을 활용해 콘텐츠를 요약하거나, Python Requests를 툴로 만들어 특정 URL의 HTML을 가져와 LLM이 분석하도록 했습니다. 이를 통해 GPT가 최신 뉴스를 읽고 핵심을 전하는 에이전트를 만들 수 있습니다.

- **유튜브 요약:** 10.3절 `youtube_summary.ipynb`에서는 유튜브 영상의 자막을 가져와 요약하는 예제를 다룹니다. 원리는 검색과 유사하게, **영상 ID로 자막 텍스트를 얻는 Tool**을 만들어 에이전트에 포함시키는 것입니다. 책의 코드에서는 `pytube`나 유튜브 API를 사용해 자막을 추출한 뒤, GPT에게 해당 자막을 요약하도록 했습니다. 결과적으로 긴 영상도 핵심만 발췌해 알려주는 에이전트를 구현할 수 있습니다.

- **스트림릿 통합:** 10.4절의 `streamlit_with_web_search.py` 등은 간단한 웹UI에서 에이전트와 상호작용하는 예제입니다. Streamlit을 사용하면 몇 줄의 코드로 입력 폼과 챗로그 출력을 만들 수 있어, 앞서 만든 검색 에이전트를 GUI로 제공할 수 있습니다. 회사 내부 포털에 이런 UI를 올려두면, 최신 정보 Q&A 봇처럼 활용할 수 있겠지요.

실습 팁: API 키와 Rate Limit

외부 정보 활용 시에는 **해당 API의 키와 쿼리 제한**을 고려해야 합니다. DuckDuckGo는 비교적 키 없이 쿼리 가능하지만, Google 검색 API, YouTube Data API 등은 키 발급과 일일 쿼리 제한이 있습니다. 실습 전 해당 API들의 사용법을 숙지하고, `.env` 등에 키를 저장한 후 `os.getenv`로 불러쓰도록 코드에 반영하세요.

또한 검색 결과를 LLM에게 넘길 때는 **토큰 비용**도 고려해야 합니다. 너무 많은 텍스트를 한번에 GPT에게 보내면 비용과 응답 시간이 커지므로, 적당히 요약하거나 필요한 부분만 발췌하는 전략이 필요합니다. LangChain의 `LLMRetrieverChain` 등을 활용하면, 검색 결과에서 Top-N 문서만 골라 LLM이 참고하게 할 수도 있습니다.

회사 내부 환경 특이사항

- **인터넷 차단:** 회사 정책상 인터넷 검색이 막혀있다면, 위 기능들은 실제로 실행되지는 않을 것입니다. 이 경우 **사내 인트라넷 검색엔진**이나 **사내 위키 문서**를 Tool로 연동하는 방향을 고려해야 합니다. 예를 들어, 사내 문서를 질의하는 API를 LangChain Tool로 만들어 두면 GPT가 외부 대신 내부 지식을 검색해 답하는 봇을 만들 수 있습니다.

- **캐싱과 로그:** 검색을 남발하면 외부 서비스 부하뿐 아니라 내부망 보안 모니터링에 로그가 잔뜩 남을 수 있습니다. 따라서 **동일 질문에 대한 캐싱**을 구현하거나, 사용 패턴을 로깅해서 비정상적인 쿼리를 감지하는 등의 운영상 대비책도 필요합니다. LangChain은 `Tracer`나 `CallbackHandler`를 통해 에이전트의 action들을 후킹할 수 있으니, 이를 활용해 어떤 검색을 얼마나 자주 호출했는지 기록해둘 수 있습니다.

다음 장에서는 OpenAI API가 아닌 **대안 LLM (오픈소스 LLM)**을 사용하는 방법과, 회사 내부 데이터를 활용하는 **RAG** 기법을 함께 살펴보겠습니다.

내부 모델과 데이터 활용: 로컬 LLM 및 RAG 기법

11장과 16장에서는 OpenAI 외의 **자체 모델을 활용하는 방법**과 **사내 데이터에 특화된 AI** 구현을 다룹니다. 이는 회사 환경처럼 민감한 데이터를 클라우드에 보내기 어려운 경우나, OpenAI API 비용을 절감하고자 할 때 특히 유용한 내용입니다.

- **로컬 LLM 사용 (예: Llama2 등):** 책의 11.2절 `deepseek_simple_chatbot.py`에서는 OpenAI 대신 **ChatOllama**를 사용한 예제가 나옵니다. 이는 서버에 올려둔 Llama 계열 14B 모델(`deepseek-r1:14b`)을 LangChain으로 호출하는 방식입니다. **Ollama**는 Llama를 API 형태로 구동시키는 도구로, `langchain_ollama.ChatOllama` 클래스가 제공됩니다. 아래는 개략적인 사용 예입니다:

```
from langchain_ollama import ChatOllama
llm = ChatOllama(model="my-llama-model")  # 로컬에 준비된 모델 지정
```

```
messages = [ SystemMessage(content="너는 친절한 도우미야.") ]
response = llm.generate(messages)
print(response.content)
```

이처럼 하면 인터넷 없이도 로컬 LLM이 답변을 생성합니다. 다만 모델 크기에 따라 서버 구동이 오래 걸리고, 응답 속도가 느릴 수 있습니다. 책의 예제에서는 대화 중 `<think>` 태그 등을 이용해 **모델의 추론 과정을 표출**하는 등 커스터마이즈도 보여주지만, 핵심은 **LangChain으로 OpenAI가 아닌 다른 LLM backend를 쓸 수 있다는 것** 입니다. HuggingFace 모델을 직접 `HuggingFacePipeline`으로 불러 사용할 수도 있습니다.

- **RAG (Retrieval Augmented Generation):** 9장과 11.3절, 그리고 13장에 걸쳐 **자체 지식 기반에 질의응답**하는 방법, 즉 RAG 기법이 다뤄집니다. RAG의 핵심은 다음과 같습니다:
- **데이터 임베딩:** 사내 문서나 도메인 지식을 문장 단위로 임베딩 벡터로 변환해 저장 (예: FAISS나 Chromadb 벡터 데이터베이스).
- **질문 임베딩 및 유사도 검색:** 사용자의 질문도 같은 방법으로 임베딩하여, 벡터DB에서 **가장 유사한 문서 조각들을 검색**.
- **LLM에 자료 제공:** 검색된 텍스트 조각들을 프롬프트에 포함시켜, GPT가 해당 정보에 기반한 답변을 생성.

책에서는 9장의 `rag_practice.ipynb`에서 서울시 기본계획과 뉴욕시 전략계획 PDF를 벡터DB에 넣고 질의응답하는 예제를 보여줍니다 (앞서 언급한 PDF 두 개입니다). 또 13장 `rag_with_langgraph.ipynb`에서는 LangGraph라는 것으로 RAG 과정을 시각화/구현했고, 11.3절 `rag_deepseek.py`에서는 **DeepSeek**이라는 사내 검색시스템 API와 RAG를 접목했습니다.

LangChain에서는 RAG를 지원하는 여러 모듈이 있는데, `VectorstoreRetriever`를 구성하고 `RetrievalQA` 체인을 쓰면 쉽게 구현 가능합니다. 예를 들어:

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

# 임베딩 모델 (예: BGE-small)
embed = HuggingFaceEmbeddings(model_name="BAAI/bge-small-ko-en")
# 로컬 문서 임베딩하여 벡터DB 생성
db = FAISS.from_texts(documents, embedding=embed)
retriever = db.as_retriever(search_kwargs={"k": 5})

from langchain.chains import RetrievalQA
qa_chain = RetrievalQA.from_chain_type(OpenAI(), retriever=retriever)
result = qa_chain.run("서울 2040 계획에서 지속가능성 언급이 나오나요?")
print(result)
```

위 코드에서 `HuggingFaceEmbeddings`를 사용한 부분이 중요합니다. 공개 한국어 임베딩 모델(BGE 등)을 활용하면 OpenAI의 유료 Embedding API 없이도 자체 벡터화가 가능합니다. 책의 16.1절 `embedding_bge_m3.ipynb`에서 바로 이 BGE 모델을 활용하는 법을 다루며, 한국어/영어 모두 잘 되는 다국어 임베딩으로 추천하고 있습니다.

- **LangGraph 소개:** 12~13장에 걸쳐 등장하는 LangGraph는 책 저자가 제안한 개념으로, **에이전트의 동작 흐름을 그래프로 시각화**하고 구성하자는 아이디어입니다. LangChain의 체인이 직렬로만 연결되는 것에서 더 나아가, 조건

에 따라 분기하거나 병렬로 수행하는 등 복잡한 로직을 다루기 위한 구조로 볼 수 있습니다. 예를 들어 RAG with LangGraph에서는 질문 -> 검색 -> 답변 생성 과정을 노드와 엣지로 표현하여, 내부 동작을 추적할 수 있게 했습니다.

LangGraph 자체는 아직 널리 쓰이는 표준은 아니지만, 해당 장을 통해 **에이전트의 사고과정을 구조적으로 생각해보는 훈련**을 할 수 있습니다. 프로젝트를 진행할 때, 처음부터 GPT에게 모든 것을 시키는 대신, 필요한 서브태스크들을 나누고 각각 검증하면서 진행하면 신뢰성을 높일 수 있다는 점을 강조합니다.

실습 팁: 임베딩 모델 및 벡터DB

임베딩 단계에서 어떤 모델을 쓰느냐에 따라 답변 정확도가 많이 좌우됩니다. 책에서는 BGE를 사용했지만, **문서의 언어와 특성에 맞는 임베딩 모델**을 고르는 것이 중요합니다. 한국어 문서 비중이 높다면 코로나임베딩(KorBERT) 등도 고려해볼 수 있고, 도메인 용어가 특수하면 해당 분야로 파인튜닝된 임베딩을 구해야 할 수도 있습니다.

벡터 데이터베이스로는 간단한 실습엔 FAISS를, 운영 환경에서는 milvus, Weaviate, ElasticSearch 등의 벡터 검색 기능을 사용할 수 있습니다. LangChain Retriever는 구현체에 관계없이 동일 인터페이스를 제공하므로, 나중에 대규모 데이터로 확장하기에도 수월합니다.

회사 내부 환경 특이사항

- **데이터 보안:** RAG를 적용할 때, 회사 내부 문서를 벡터화하여 외부 LLM API에 보낼 경우 **민감 정보 유출 위험**이 있습니다. 벡터는 원본을 재생성하기 어렵다고 하지만, GPT에 비밀 텍스트를 통으로 넣는 것은 위험할 수 있습니다. 이를 완화하려면 사내 문서 중 공개 가능한 범위의 자료만 사용하거나, 아예 **사내 전용 LLM**을 활용하는 것이 좋습니다. 11장의 DeepSeek 예제처럼, 사내용 LLM (`ChatOllama` 등)과 사내 검색엔진을 조합하면 클라우드에 내용이 나가지 않으므로 안전성을 높일 수 있습니다.

- **연산 자원:** 로컬 LLM(특히 수십억 파라미터)과 임베딩 계산은 모두 **GPU 자원**을 요합니다. 회사에서 이러한 프로젝트를 진행할 때는, 중앙 GPU 서버나 AI 전용 장비를 할당받아 거기서 서비스를 돌리는 형태가 필요합니다. 개발 단계에서는 데스크톱 GPU나 Colab 등을 쓰겠지만, 운영 시에는 컨테이너로 배포해 API 형태로 제공하는 것을 고려해야 합니다. 책의 후반 프로젝트도 Streamlit 데모 수준이지만, 실제 서비스로 확장하려면 Flask/FastAPI 백엔드 등으로 리팩터링이 필요할 것입니다.

마지막으로, 14~15장에 걸쳐 진행되는 **종합 프로젝트 예제**를 살펴보고 마무리하겠습니다.

종합 프로젝트: 책 작성 에이전트 (멀티에이전트 협업)

책의 마지막 부분은 지금까지 배운 기능들을 총동원하여 **"AI 책쓰기 도우미"** 에이전트를 만드는 프로젝트입니다. 이 프로젝트는 다양한 역할의 GPT 에이전트들이 협업하여 최종 산출물을 만들어내는 구조로, 일종 of **멀티에이전트 시스템**의 예시입니다.

- **시나리오:** 사용자(작가)가 책의 개요(outline)를 입력하면, AI 에이전트들이 챕터별로 내용을 작성하고 검토하여 최종 원고를 완성하도록 구성됩니다. 이를 위해 **Content Strategist**(구성 전략가), **Communicator**(텍스트 작성자), **Research Agent**(자료 조사), **Critic/Reviewer**(검토자) 등의 역할을 부여한 여러 GPT 인스턴스를 사용합니다. 이들은 서로 역할에 맞는 프롬프트로 대화하며 점진적으로 콘텐츠를 다듬어 갑니다. 책에 첨부된 다이어그램에서도 이러한 역할 구조가 나타나 있습니다 (content_strategist, communicator, vector_search_agent, supervisor 등 노드로 표현)【22†output】【22†output】.

- **단계별 개발:** 14장의 각 절과 15장의 각 절마다 기능을 점진적으로 추가합니다. 예를 들면:

- 14.1절: 기본적인 책 챕터 생성 흐름 구현. (한 챕터 분량의 텍스트 생성)
- 14.2절: 외부 자료 검색 에이전트 통합. (주제와 관련된 정보를 벡터 검색으로 보강)
- 14.3절: 도구 사용 및 다중 단계 체인. (예: 웹 검색 에이전트를 활용하여 최신 정보도 반영)
- 14.4절: 에이전트 간 스트림 통신 구현. (각 역할의 intermediate output을 주고받으며 실시간 협업하는 형태)
- 15.1~15.3절: GUI 개선, 프롬프트 미세튜닝, 전체 시스템 테스트 및 개선.

개발자 입장에서 이러한 **Iterative Development** 과정은 복잡한 LLM 어플리케이션을 만들 때 매우 중요합니다. 처음부터 거대한 프롬프트 하나로 원하는 결과를 얻기보다는, 부분적으로 역할을 나눠 조율하는 접근이 효과적임을 보여줍니다.

- **예제 코드 분석:** 프로젝트의 핵심은 `book_writer.py` 스크립트들로, 내부에 각 역할별 에이전트 프롬프트와 동작이 정의되어 있습니다. 예를 들어, **content_strategist_chain**은 입력된 책 개요를 기반으로 각 장의 세부 주제를 생성하고, **communicator_chain**은 그 주제에 대한 본문을 작성합니다. 그런 다음 **supervisor** 역할의 에이전트가 초안을 검토해 수정사항을 지시하면, communicator가 반영하는 식입니다. 이 모든 흐름을 파이썬 코드로 일일이 orchestrate하고 있으며, LangChain보다는 저자가 만든 LangGraph 스타일로 구현되어 있습니다.

코드 상에서 흥미로운 부분은 파이프 연산자 `|`를 오버로딩하여 `prompt | llm | parser` 식으로 체인을 표현한 부분인데, 이는 가독성을 위해 한 시도이며 본질적으로 `LLMChain`과 비슷한 동작을 합니다 [1]. 또한 `<think>` 태그 등을 이용해 에이전트의 *생각(내부 독백)*과 최종 산출을 구분하는 프롬프트 기법도 등장합니다. 이런 기법들은 멀티에이전트 협업 시, 각각의 에이전트가 혼동 없이 일하도록 유도하는 데 활용됩니다.

- **최종 출력:** 사용자가 한 문단의 개요만 제공해도, 여러 GPT가 상호작용하며 수십 페이지 분량의 초안을 생성해내는 것이 목표입니다. 당연히 완벽한 책 수준은 아니고 사람이 다듬어야 하지만, **창작 보조 AI**로서의 가능성을 엿볼 수 있습니다.

실습 팁: 프롬프트 엔지니어링과 실패 대처

멀티에이전트 시스템에서는 한 에이전트의 출력이 다른 에이전트의 입력으로 들어가기 때문에, **프롬프트 엔지니어링의 복잡도가 배가**됩니다. 각 단계의 에이전트가 정확히 자기 역할만 수행하도록 지시하고, 불필요한 산출을 내놓지 않도록 컨트롤해야 합니다. 예를 들어, "Research Agent"는 웹에서 찾은 정보만 전달하고 의견은 첨언하지 말라든지, "Critic"은 오탈자나 일관성만 체크하고 내용 수정은 하지 말라든지 세세한 규칙을 정해야 합니다.

그럼에도 불구하고 LLM의 출력은 가변성이 크므로, **예상치 못한 포맷이나 협업 오류**가 생길 수 있습니다. 이런 경우 로그를 면밀히 살펴보고, 특정 에이전트의 프롬프트에 추가 제약을 넣어 해결해야 합니다. 책의 예제 코드에서도 여러 차례 수정을 거쳤음을 알 수 있는데, 이러한 **디버깅 과정**은 LLM 앱 개발의 일상임을 염두에 두고 인내심을 갖고 조율해야 합니다.

회사 내부 환경 특이사항

- **모델 리소스:** 멀티에이전트라 해서 GPT API를 여러 번 병렬 호출하면 비용과 시간이 많이 들 수 있습니다. 사내에서 운영한다면, 꼭 필요한 에이전트만 두고 나머지는 비활성화하거나, 한 번의 API호출에 여러 역할을 같이 수행하도록 프롬프트를 최적화하는 등 **효율화**가 필요합니다. 예를 들어 컨텐츠 전략가와 작성자를 한 프롬프트 내 단계로 묶고, 검토자를 별도로 두는 2스텝으로 단순화한다든지 하는 식입니다.

- **특정 도메인 적용:** 책쓰기 에이전트 예제를 회사 상황에 맞게 변형한다면, 예컨대 **보고서 작성 보조 AI**로 만들 수 있을 것입니다. 이 때는 회사의 내부 지식베이스를 Research Agent가 활용하도록 하고, 어조나 형식을 사내 양식에 맞추도록 프롬프트를 설계해야 합니다. LLM에게 특정 사내 용어를 사용하게 하거나 금칙어를 피하게 하는 규칙도

필요할 수 있습니다. 이러한 **도메인 적응 작업** 역시 많은 테스트와 수정이 수반되지만, LangChain이나 기존 예제 코드를 기반으로 하면 개발 생산성을 높일 수 있습니다.

결론 및 추가 학습 자료

이 가이드를 통해 **대규모 언어모델(LLM) 활용 AI 에이전트** 개발의 주요 개념과 구현 방식을 살펴보았습니다. 요약하면:

- **프롬프트 설계와 OpenAI API 기본 활용**으로 시작하여, 텍스트 요약, 음성 인식, 이미지 캡션 생성 등 **멀티모달 입력 처리**를 거쳤습니다.
- GPT의 **함수 호출 기능**과 LangChain의 **체인, 에이전트** 기능을 통해 LLM의 한계를 보완하고 외부 도구와 연계하는 방법을 배웠습니다.
- 나아가 **로컬 LLM 및 벡터DB를 이용한 RAG**로 사내 데이터에 특화된 QA 시스템을 만들고, **복수의 LLM 에이전트가 협업**하는 고급 주제도 다뤘습니다.

학습을 진행하면서 반드시 **실습 코드**를 직접 실행해보고, 필요에 따라 프롬프트나 파라미터를 바꿔 결과가 어떻게 달라지는지 관찰해보세요. LLM 기반 개발은 완전히 동일한 코드를 실행해도 매번 결과가 다를 수 있으므로, **여러 시행착오를 통해 직관을 기르는 것**이 중요합니다.

마지막으로, 혼자 공부를 마친 후에도 아래 자료를 참고하여 꾸준히 지식을 업데이트하시길 추천합니다:

- 최신 LangChain 공식 문서 및 예제 (버전 변화가 잦으므로 공식 가이드 확인)
- Hugging Face Hub의 새로운 한국어 LLM/임베딩 모델 공개 동향
- OpenAI 함수 호출 및 Tool 사용 예제 (OpenAI Cookbook 등에 사례 다수)
- 멀티에이전트 AI에 관한 연구 커뮤니티 (Self-Reflective LLM, Auto-GPT류 프로젝트 등)

LLM 기술은 나날이 발전하고 있습니다. **환경 제한이 있더라도 창의적인 방법으로** 이를 활용하여 업무 생산성을 높일 수 있는 길은 많습니다. 이번 가이드가 그 첫걸음이 되길 바라며, 직접 작동하는 멋진 AI 에이전트를 만들어보시기 바랍니다!

---

[1] Do it! LLM을 활용한 AI 에이전트 개발 입문 정오표 - Hojas de cálculo de Google
https://docs.google.com/spreadsheets/d/123h6qq5SNDtp4WVTvVbloFosAX3DRIALrkoFKb7MMSg/edit?gid=0