CH04 - 모델(Model)

LLM (Large Language Model) 단계

개요

모델 혹은 **LLM(Large Language Model)** 단계는 **이전 프롬프트 단계에서 구성된 입력을 기반으로**, 대규모 언어 모델을 활용해 **응답을 생성하는 과정**이다.

이 단계는 **RAG 시스템의 핵심**으로, 언어 모델의 능력을 활용해 **정확하고 자연스러운 답변**을 생성한다.

LLM의 필요성

1. **사용자 의도 이해 (NLU + NLG)**
    - LLM은 언어의 구조와 의미를 깊이 이해하며, 복잡한 질문에도 논리적인 답변을 제공한다.
    - *자연어 이해(NLU)**와 **자연어 생성(NLG)** 능력이 결합되어, **자연스럽고 유익한 응답**을 생성한다.
2. **문맥적 적응성 (Context Awareness)**
    - LLM은 주어진 문맥을 고려해 답변을 생성한다.
    - 사전학습된 지식뿐 아니라 **사용자가 제공한 추가 정보**를 기반으로 문맥을 반영한 응답을 생성한다.

LLM의 중요성

- LLM 단계는 **답변의 질과 자연스러움을 결정짓는 핵심 요소**이다.
- 지금까지의 모든 데이터와 정보를 종합해 **최적화된 답변**을 생성한다.
- LLM의 성능은 RAG 시스템 전체의 효율성과 **사용자 만족도**에 직접적인 영향을 미친다.
- 따라서 **LLM은 RAG 시스템의 중심 엔진**이라 할 수 있다.

코드 예시

OpenAI GPT-4o 활용

```
# 단계 7: 언어모델(LLM) 생성
# OpenAI의 GPT-4o 모델을 생성합니다.
llm = ChatOpenAI(model_name="gpt-4o")
```

Anthropic Claude 3 Sonnet 활용

```
from langchain_anthropic import ChatAnthropic

# 단계 7: 언어모델(LLM) 생성
# Anthropic의 Claude 모델을 생성합니다.
llm = ChatAnthropic(model="claude-3-sonnet-20240229")
```

로컬 모델 (Llama3-8B) 활용

```
from langchain_community.chat_models import ChatOllama

# 단계 7: 언어모델(LLM) 생성
# LangChain이 지원하는 Ollama(로컬) 모델을 사용합니다.
llm = ChatOllama(model="llama3:8b")
```

정리 포인트

구분	주요 역할	예시 모델
OpenAI	GPT-4o 기반 LLM	`ChatOpenAI`
Anthropic	Claude 3 Sonnet	`ChatAnthropic`
Local	Llama3 (Ollama 기반)	`ChatOllama`

I. 다양한 LLM 모델 활용

1️⃣ OpenAI

- **대표 모델:** GPT-4o, GPT-4-turbo, GPT-4o-mini, o1-preview, o1-mini
- **특징:** 높은 정확도, 다중모달 지원(텍스트·이미지·음성), 다양한 옵션 (`temperature`, `max_tokens`)

- **코드 예시:**

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
```

2️⃣ Anthropic (Claude)

- **대표 모델:** Claude 3.5 Sonnet, Claude 3 Opus, Claude 3 Haiku
- **특징:** 윤리적·안전성 중심의 대규모 언어모델, 논리적 응답 강점
- **코드 예시:**

```
from langchain_anthropic import ChatAnthropic
llm = ChatAnthropic(model_name="claude-3-5-sonnet-20241022")
```

3️⃣ Perplexity

- **대표 모델:** Llama 3.1 Sonar (Small / Large / Huge)
- **특징: 검색 기반 LLM**으로, 실시간 인용(citations) 제공
- **기능:** PDF·이미지 분석, 관련 질문 추천, API 제공
- **코드 예시:**

```
from langchain_teddynote.models import ChatPerplexity
llm = ChatPerplexity(model="llama-3.1-sonar-large-128k-online", temper
ature=0.2)
```

4️⃣ Cohere

- **대표 모델:** Command R+, Aya (다국어 지원 오픈소스)
- **특징:** 기업용 AI 솔루션, 128k 토큰 지원, 고급 RAG 기능 탑재
- **코드 예시:**

```
from langchain_cohere import ChatCohere
llm = ChatCohere(temperature=0)
```

5️⃣ Upstage

- **대표 모델:** Solar LLM
- **특징:** 한국 스타트업 기반, 빠른 성능·비용 효율성, Document AI 강점
- **코드 예시:**

```
from langchain_upstage import ChatUpstage
llm = ChatUpstage(model="solar-pro")
```

6️⃣ Xionic (사이오닉 AI)

- **대표 모델:** xionic-1-72b
- **특징:** 한국어 특화 상용 모델, 기업용 생성형 AI
- **코드 예시:**

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(
    model_name="xionic-1-72b-20240919",
    base_url="https://sionic.chat/v1/",
    api_key="API_KEY",
)
```

7️⃣ LogicKor

- **정의:** 한국어 LLM 성능을 평가하는 **벤치마크 리더보드**
- **평가 영역:** 추론, 수학, 글쓰기, 코딩, 이해력
- **목적:** 국내외 모델의 사고력 비교 및 객관적 평가 제공

II. 캐싱(Cahce)

개요

LangChain은 **LLM 응답을 캐싱(Cache)** 하여

- **API 호출 비용을 절감**하고
- **응답 속도를 향상**시키기 위한 캐시 레이어를 제공한다.

동일한 입력에 대해 반복 호출할 때,

이전 결과를 저장하고 **즉시 재사용**할 수 있다.

주요 캐시 방식

1️⃣ InMemoryCache (메모리 기반)

- **특징:** 실행 중 메모리에만 저장되며, 프로그램 종료 시 사라짐
- **장점:** 빠른 속도, 간단한 설정
- **예시:**

```
from langchain.globals import set_llm_cache
from langchain.cache import InMemoryCache

set_llm_cache(InMemoryCache())
```

2️⃣ SQLiteCache (파일 기반)

- **특징:** `SQLite` 데이터베이스에 캐시를 저장 (지속형)
- **장점:** 프로그램을 껐다 켜도 캐시 유지
- **예시:**

```
from langchain_community.cache import SQLiteCache
from langchain_core.globals import set_llm_cache

set_llm_cache(SQLiteCache(database_path="cache/llm_cache.db"))
```

사용 흐름 요약

1. `set_llm_cache()` 로 캐시 타입 지정
2. LLM 호출 시 결과가 자동 저장
3. 동일한 입력 시 캐시된 결과 즉시 반환

효과 비교 요약

구분	저장 위치	속도	지속성	적합한 경우
InMemoryCache	RAM	매우 빠름	❌ 비지속	단기 테스트용
SQLiteCache	로컬 파일	빠름	✅ 지속	장기 프로젝트용

III. 모델 직렬화 (Serialization)

개요

- *직렬화(Serialization)**는 모델이나 객체를 **저장 가능한 형식으로 변환**하는 과정이다.

AI 개발에서 **모델 재사용, 배포, 버전 관리, 리소스 절약**을 위해 필수적인 단계이다.

목적

- 모델 **재훈련 없이 재사용** 가능
- **배포 및 공유 용이**
- **저장 공간 및 계산 리소스 절약**

장점

- 빠른 **모델 로딩**
- **버전 관리** 용이
- **다양한 환경**에서 동일 모델 사용 가능

직렬화 가능 여부 확인

LangChain 클래스나 객체가 직렬화 가능한지 확인할 수 있다.

```
from langchain_openai import ChatOpenAI
print(ChatOpenAI.is_lc_serializable())  # True
```

체인(Chain) 직렬화 방법

1️⃣ `dumpd()` — 딕셔너리 형태로 직렬화

```
from langchain_core.load import dumpd
dumpd_chain = dumpd(chain)
```

```
type(dumpd_chain)  # dict
```

2️⃣ `dumps()` — JSON 문자열 형태로 직렬화

```
from langchain_core.load import dumps
dumps_chain = dumps(chain)
type(dumps_chain)  # str
```

Pickle 직렬화

특징

구분	내용
형식	Python 객체를 **바이너리 형태**로 저장
장점	빠른 직렬화, 복잡한 객체 구조 유지
단점	**Python 전용**, 보안 주의 필요
주요 용도	객체 캐싱, 모델 저장, 프로그램 상태 복원

사용 예시

```
import pickle

# 저장
with open("fruit_chain.pkl", "wb") as f:
    pickle.dump(dumpd_chain, f)

# 불러오기
with open("fruit_chain.pkl", "rb") as f:
    loaded_chain = pickle.load(f)
```

JSON 직렬화

```
import json

# 저장
with open("fruit_chain.json", "w") as fp:
```

```
    json.dump(dumpd_chain, fp)

# 로드
with open("fruit_chain.json", "r") as fp:
    loaded_json = json.load(fp)
```

LangChain에서 로드하기

```
from langchain_core.load import load, loads

# Pickle 기반 로드
chain_from_file = load(loaded_chain)

# JSON 기반 로드
loads_chain = load(loaded_json)

# 테스트 실행
loads_chain.invoke({"fruit": "사과"})
```

요약 표

구분	저장 형식	특징	장점	비고
dumpd	dict	파이썬 내 데이터 구조 저장	간단한 직렬화	메모리 기반
dumps	JSON 문자열	텍스트 저장용	파일 전송/공유 용이	
Pickle	바이너리	Python 전용 고속 직렬화	빠른 저장·복원	보안 주의
JSON	텍스트	언어 간 호환 가능	가독성 높음	용량 큼

IV. 토큰 사용량 확인

토큰 사용량 확인 (Token Usage Tracking)

개요

LangChain은 **OpenAI 모델 호출 시 토큰 사용량과 요금**을 추적할 수 있다.

이 기능은 **현재 OpenAI API 전용**으로 지원된다.

목적

- 모델 호출별 **토큰 소비량** 및 **비용 확인**
- **프롬프트·응답 토큰 구분**
- 모델 사용 최적화 및 **비용 관리**

사용 방법

1️⃣ 콜백(callback)으로 추적

`get_openai_callback()` 을 이용하면

`with` 블록 내부의 모든 LLM 호출에 대해 토큰 정보를 자동 수집한다.

```
from langchain.callbacks import get_openai_callback
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model_name="gpt-4o")

with get_openai_callback() as cb:
    result = llm.invoke("대한민국의 수도는 어디야?")
    print(cb)
```

출력 예시

```
Tokens Used: 55
Prompt Tokens: 15
Completion Tokens: 40
Successful Requests: 1
Total Cost (USD): $0.00067
```

2️⃣ 여러 호출에 대한 누적 추적

```
with get_openai_callback() as cb:
    result = llm.invoke("대한민국의 수도는 어디야?")
```

```
result = llm.invoke("서울은 어떤 도시야?")
print(f"총 토큰: {cb.total_tokens}")
print(f"프롬프트 토큰: {cb.prompt_tokens}")
print(f"응답 토큰: {cb.completion_tokens}")
print(f"총 비용(USD): ${cb.total_cost}")
```

주요 출력 항목

항목	설명
total_tokens	전체 토큰 수
prompt_tokens	입력(프롬프트)에 사용된 토큰 수
completion_tokens	모델 출력(응답)에 사용된 토큰 수
total_cost	총 사용 금액(USD)
successful_requests	성공적으로 수행된 요청 횟수

활용 포인트

- 모델의 **비용 효율성 분석**
- **대화형 시스템 최적화** (불필요한 토큰 절약)
- **프로젝트별 요금 관리** 및 모니터링 가능

V. 구글 생성 AI (Google Generative AI)

개요

Google의 Gemini 시리즈는 텍스트·이미지 등 멀티모달 입력을 지원하는 생성형 AI 모델이다.

LangChain에서는 `ChatGoogleGenerativeAI` 클래스를 통해 접근할 수 있다.

주요 모델

모델명	설명
gemini-1.5-pro-latest	고성능 멀티모달 모델 (텍스트·이미지 입력 지원)
gemini-1.5-flash-latest	빠르고 가벼운 버전 (비용 효율 중심)

모델명	설명
gemini-vision	이미지 중심 인식 및 설명용 모델

기본 사용법

```
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro-latest")
answer = llm.stream("자연어처리에 대해 간략히 설명해줘")
```

프롬프트 체인 예시

```
from langchain_core.prompts import PromptTemplate

prompt = PromptTemplate.from_template("{question}는 과일입니까?")
chain = prompt | ChatGoogleGenerativeAI(model="gemini-1.5-flash-latest")

chain.stream({"question": "사과"})
```

Safety Settings (안전 설정)

모델의 콘텐츠 차단 수준을 설정할 수 있다.

```
from langchain_google_genai import ChatGoogleGenerativeAI, HarmBlockTh
reshold, HarmCategory

llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-pro-latest",
    safety_settings={
        HarmCategory.HARM_CATEGORY_HATE_SPEECH: HarmBlockThreshol
d.BLOCK_NONE,
        HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT: HarmBlockThr
eshold.BLOCK_NONE,
    },
)
```

> ⚙️ 필요에 따라 HarmBlockThreshold 값을 조정해 “Safety Warnings”을 줄일 수 있음.

Batch 실행

여러 입력을 한 번에 처리할 수 있다.

```
llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro-latest")
results = llm.batch(["대한민국의 수도는?", "주요 관광지 5곳을 알려줘"])
```

멀티모달 (Multimodal) 활용

텍스트 + 이미지 입력을 동시에 처리 가능.

```
from langchain_teddynote.models import MultiModal

gemini = ChatGoogleGenerativeAI(model="gemini-1.5-pro-latest")
multimodal = MultiModal(gemini,
    system_prompt="당신은 시인입니다.",
    user_prompt="이미지를 보고 시를 작성해주세요."
)
answer = multimodal.stream("images/jeju-beach.jpg")
```

요약 정리

구분	기능	특징
텍스트 생성	대화형 응답	자연스러운 문장 생성
Batch 실행	여러 프롬프트 동시 처리	속도 향상
Safety 설정	위험 콘텐츠 제어	유해 표현 차단 가능
멀티모달 지원	이미지·텍스트 입력 가능	창의적 응용 가능

VI. 허깅페이스 엔드포인트(HuggingFace Endpoints)

개요

Hugging Face Hub는

- **12만+ 모델**, **2만+ 데이터셋**, **5만+ Spaces(데모 앱)**을 보유한 **오픈소스 AI 플랫폼**이다.
- 모델 공유, 협업, 배포가 가능하며 **Inference API**와 **Inference Endpoints**를 통해 모델을 직접 호출할 수 있다.

주요 특징

- **오픈소스 중심 생태계** (모델/데이터/앱 공유)
- **Inference API:** 무료·서버리스 예측 서비스
- **Inference Endpoints:** 전용 인프라 기반의 프로덕션 서비스
- **빠른 추론 엔진:** Rust·Python·gRPC 기반 **Text Generation Inference**

토큰 발급

1. huggingface.co 가입
2. 토큰 발급 페이지 접속
3. `.env`에 토큰 저장

```
HUGGINGFACEHUB_API_TOKEN=발급받은_토큰
```

기본 사용법

```
from langchain_huggingface import HuggingFaceEndpoint
from langchain.prompts import PromptTemplate

prompt = PromptTemplate.from_template("{question}")

llm = HuggingFaceEndpoint(
    repo_id="microsoft/Phi-3-mini-4k-instruct",
    max_new_tokens=256,
    temperature=0.1,
    huggingfacehub_api_token=os.environ["HUGGINGFACEHUB_API_TOKEN"],
)

response = llm.invoke("What is the capital of South Korea?")
```

```
print(response)
# ➜ Seoul
```

엔드포인트 유형

구분	이름	특징	용도
Serverless Endpoint	Inference API	무료, 간단한 실험용	테스트 및 프로토타입
Dedicated Endpoint	Inference Endpoints	전용 서버·AutoScaling·보안 지원	대규모 서비스 운영

전용 엔드포인트 예시

```
llm = HuggingFaceEndpoint(
    endpoint_url="https://xxxx.aws.endpoints.huggingface.cloud",
    max_new_tokens=512,
    temperature=0.01,
)
llm.invoke("대한민국의 수도는 어디인가요?")
```

참고 자료

- 🔗 Hugging Face LLM 리더보드
- 🔗 모델 목록
- 🔗 LogicKor 리더보드

VII. 허깅페이스 로컬(HuggingFace Local)

Hugging Face Pipeline (로컬 모델 사용)

개요

`HuggingFacePipeline`은 **허깅페이스 모델을 직접 로컬에 다운로드하여 실행**할 수 있는 방식이다.

인터넷 연결 없이도 빠른 추론이 가능하며, 모델을 **캐시 디렉토리**에 저장해 반복 다운로드를 방지한다.

설정 방법

1️⃣ 모델 다운로드 경로 설정

```
import os

# 모델 및 토크나이저 다운로드 캐시 경로 지정
os.environ["TRANSFORMERS_CACHE"] = "./cache/"
os.environ["HF_HOME"] = "./cache/"
```

2️⃣ 모델 불러오기

```
from langchain_huggingface import HuggingFacePipeline

llm = HuggingFacePipeline.from_model_id(
    model_id="microsoft/Phi-3-mini-4k-instruct",
    task="text-generation",
    pipeline_kwargs={
        "max_new_tokens": 256,
        "top_k": 50,
        "temperature": 0.1,
    },
)
```

✅ **예시 모델:** microsoft/Phi-3-mini-4k-instruct

예제 1: 기본 텍스트 생성

```
llm.invoke("Hugging Face is")
```

→ 모델이 간단한 설명 문장을 생성함.

예제 2: 프롬프트 체인과 연결

```
from langchain_core.prompts import PromptTemplate

template = """Summarize TEXT in bullet points from most important to least important.
TEXT:
{text}

KeyPoints:"""

prompt = PromptTemplate.from_template(template)
chain = prompt | llm

response = chain.invoke({"text": "A Large Language Model (LLM) like ChatGPT..."})
print(response)
```

출력 예시

```
- LLMs are AI models that understand, generate, and interact with human language.
- They predict the next word to produce coherent text.
- Used for Q&A, writing, and code generation.
- Trained on diverse text sources.
- Raise concerns on bias, ethics, and accuracy.
```

장점

항목	설명
로컬 실행 가능	인터넷 없이 추론 가능
빠른 응답 속도	캐시 사용으로 성능 향상
비용 절감	API 호출 비용 없음
유연한 프롬프트 연결	LangChain 체인과 쉽게 통합 가능

VIII. 허깅페이스 파이프라인(HuggingFace Pipeline)

Hugging Face Local Pipelines

개요

`HuggingFacePipeline`은 **허깅페이스 모델을 로컬 환경에서 직접 실행**할 수 있도록 하는 LangChain 래퍼 클래스이다.

모델은 Hugging Face Model Hub에서 다운로드되어 **캐시 디렉토리**에 저장되고, 이후 오프라인에서도 사용할 수 있다.

> ⚙️ 필수 패키지: transformers, torch
>
> (선택) 고성능 메모리 효율화를 위해 `xformers` 설치 가능

1️⃣ 모델 다운로드 경로 설정

```
import os

# 모델과 토크나이저 다운로드 경로 설정
os.environ["TRANSFORMERS_CACHE"] = "./cache/"
os.environ["HF_HOME"] = "./cache/"
```

2️⃣ 모델 로드 (from_model_id 사용)

```
from langchain_community.llms.huggingface_pipeline import HuggingFacePi
peline

hf = HuggingFacePipeline.from_model_id(
    model_id="beomi/llama-2-ko-7b",
    task="text-generation",
    pipeline_kwargs={"max_new_tokens": 512},
)
```

✅ **참고 모델:** beomi/llama-2-ko-7b

3️⃣ 직접 파이프라인 구성 (transformers 이용)

```
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
from langchain_community.llms.huggingface_pipeline import HuggingFacePi
```

```
peline

model_id = "beomi/llama-2-ko-7b"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id)

pipe = pipeline("text-generation", model=model, tokenizer=tokenizer, max_n
ew_tokens=512)
hf = HuggingFacePipeline(pipeline=pipe)
```

4️⃣ 프롬프트 체인 구성

```
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser

template = """Answer the following question in Korean.
#Question:
{question}

#Answer:"""

prompt = PromptTemplate.from_template(template)
chain = prompt | hf | StrOutputParser()

print(chain.invoke({"question": "대한민국의 수도는 어디야?"}))
```

5️⃣ GPU Inference

```
gpu_llm = HuggingFacePipeline.from_model_id(
    model_id="beomi/llama-2-ko-7b",
    task="text-generation",
    device=0,  # GPU ID (0부터 시작)
    pipeline_kwargs={"max_new_tokens": 64},
)
```

```
gpu_chain = prompt | gpu_llm | StrOutputParser()
print(gpu_chain.invoke({"question": "대한민국의 수도는 어디야?"}))
```

> 💡 device=-1 → CPU,
>
> `device_map="auto"` → 여러 GPU 자동 분배 (Accelerate 필요)

6️⃣ Batch GPU Inference

```
gpu_llm = HuggingFacePipeline.from_model_id(
    model_id="beomi/llama-2-ko-7b",
    task="text-generation",
    device=0,
    batch_size=2,
    model_kwargs={"temperature": 0, "max_length": 256},
)

gpu_chain = prompt | gpu_llm.bind(stop=["\n\n"])

questions = [{"question": f"숫자 {i} 이 한글로 뭐에요?"} for i in range(4)]
answers = gpu_chain.batch(questions)

for ans in answers:
    print(ans)
```

요약 표

구분	설명
클래스	`HuggingFacePipeline`
모델 실행 위치	로컬 환경 (CPU/GPU)
필요 패키지	`transformers`, `torch`
장점	빠른 추론, 오프라인 실행, 비용 절감
GPU 지원	`device=0`, `device_map="auto"`
배치 처리	`.batch()` 로 병렬 질의 가능

IX. 올라마(Ollama)

Ollama (로컬 오픈소스 LLM 실행)

개요

Ollama는 Llama 3 등 오픈소스 대규모 언어 모델을 **로컬에서 GPU를 활용해 실행**할 수 있는 플랫폼이다.

모델은 **Modelfile**로 정의되어 있으며, 가중치·설정·데이터를 하나의 패키지로 관리한다.

LangChain과 연동해 완전한 로컬 LLM 환경을 구성할 수 있다.

🔗 설치 주소

🔗 Ollama Model Library

1️⃣ 설치 및 모델 다운로드

■ 설치

Mac / Linux / Windows에서 Ollama를 다운로드 후 설치

■ 모델 다운로드 명령어

```
ollama pull gemma:7b
ollama list      # 설치된 모델 목록 확인
ollama run llama3:8b  # CLI에서 직접 실행
```

■ 기본 모델 저장 경로

OS	경로
Mac	`~/.ollama/models`

OS	경로
Linux / WSL	`/usr/share/ollama/.ollama/models`

2️⃣ Modelfile 예시

```
FROM ggml-model-Q5_K_M.gguf

TEMPLATE """{{- if .System }}
<s>{{ .System }}</s>
{{- end }}
<s>Human:
{{ .Prompt }}</s>
<s>Assistant:
"""

SYSTEM """A chat between a curious user and an artificial intelligence assis
tant."""
PARAMETER stop <s>
PARAMETER stop </s>
```

> 💡 Modelfile을 이용하면 모델 설정, 시스템 프롬프트, 토큰 규칙 등을 직접 커스터마이징 가능

3️⃣ LangChain + Ollama 연동

```
from langchain_community.chat_models import ChatOllama
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_teddynote.messages import stream_response

llm = ChatOllama(model="EEVE-Korean-10.8B:latest")
prompt = ChatPromptTemplate.from_template("{topic} 에 대해 간략히 설명해
줘.")

chain = prompt | llm | StrOutputParser()
answer = chain.stream({"topic": "deep learning"})
stream_response(answer)
```

예시 출력:

> 딥러닝은 인간의 뇌를 모방한 인공 신경망을 통해 대규모 데이터를 학습하는 기계학습의 한 분야입니다...

4️⃣ 비동기 스트리밍 (Async Streaming)

```
async for chunk in chain.astream({"topic": "Google"}):
    print(chunk, end="", flush=True)
```

> 토큰 단위로 실시간 출력 가능 (비동기 방식)

5️⃣ JSON 형식 응답

```
llm = ChatOllama(
    model="gemma:7b",
    format="json",
    temperature=0,
)

prompt = "유럽 여행지 10곳을 알려주세요. key: `places`. response in JSON form
at."
response = llm.invoke(prompt)
print(response.content)
```

📄 **예시 출력**

```
{
  "places": [
    "프랑스의 파리",
    "이탈리아의 로마",
    "스페인의 바르셀로나",
    "스위스의 제네바",
    "스코틀랜드의 에든버러"
  ]
}
```

6️⃣ 멀티모달 (Multimodal) 지원

Ollama는 **bakllava**, **llava** 등 이미지+텍스트 입력을 지원하는 모델을 제공한다.

`PIL → Base64` 변환을 통해 이미지를 프롬프트에 포함할 수 있다.

```
from langchain_community.chat_models import ChatOllama
from langchain_core.messages import HumanMessage
from langchain_core.output_parsers import StrOutputParser
from PIL import Image
import base64, io

def convert_to_base64(pil_image):
    buffered = io.BytesIO()
    pil_image.save(buffered, format="JPEG")
    return base64.b64encode(buffered.getvalue()).decode("utf-8")

def prompt_func(data):
    text, image = data["text"], data["image"]
    return [HumanMessage(content=[
        {"type": "image_url", "image_url": f"{image}"},
        {"type": "text", "text": text},
    ])]

img = Image.open("./images/jeju-beach.jpg")
image_b64 = convert_to_base64(img)

llm = ChatOllama(model="llava:7b", temperature=0)
chain = prompt_func | llm | StrOutputParser()

result = chain.invoke({"text": "Describe this image", "image": image_b64})
print(result)
```

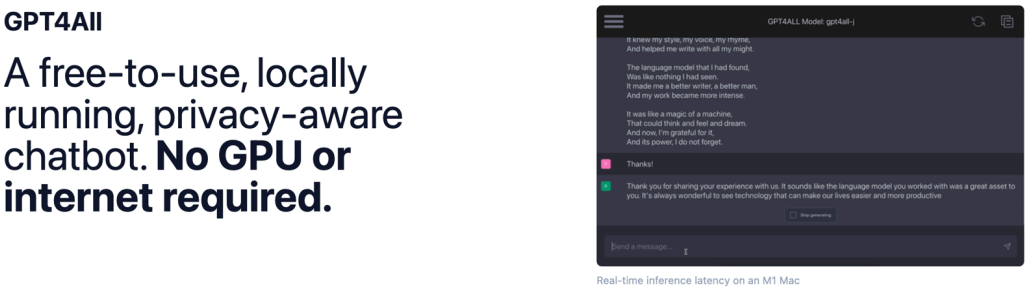
출력 예시

```
- A clear blue beach with waves breaking on the rocks.
- A distant island in the background.
- Bright weather and calm scenery.
```

요약 표

구분	설명
플랫폼	Ollama
실행 위치	로컬 (GPU/CPU)
지원 모델	Llama, Gemma, EEVE, Llava 등
입출력 형식	Text / JSON / Multimodal
장점	로컬 추론, 프라이버시 보장, 실시간 스트리밍
통합 지원	LangChain `ChatOllama`

X. GPT4ALL



GPT4All (로컬 오픈소스 챗봇)

개요

GPT4All은 nomic-ai에서 개발한 오픈소스 챗봇 생태계로,

코드·대화·지식 데이터를 포함한 방대한 텍스트로 학습된 **로컬 실행형 LLM**이다.

> GPT-3와 유사한 구조의 모델이며, 완전히 오프라인 환경에서도 작동 가능

🔗 공식 GitHub: nomic-ai/gpt4all

1️⃣ 설치

■ 프로그램 설치

공식 홈페이지에서 OS에 맞는 실행 파일 다운로드 후 설치

👉 GPT4All 공식 사이트

■ Python 패키지 설치

```
pip install -qU gpt4all
```

2️⃣ 모델 다운로드

GPT4All **Model Explorer**에서 모델을 선택하고 다운로드한다.

> ⚠️ 본인 PC 사양(GPU 메모리, RAM 등)에 맞는 모델을 선택해야 함.

예시 모델:

`EEVE-Korean-Instruct-10.8B-v1.0-Q8_0.gguf`

```
mkdir models
# 모델 파일을 ./models 폴더에 저장
```

로컬 경로 지정:

```
local_path = "./models/EEVE-Korean-Instruct-10.8B-v1.0-Q8_0.gguf"
```

3️⃣ LangChain 연동

```
from langchain.prompts import ChatPromptTemplate
from langchain_community.llms import GPT4All
from langchain_core.output_parsers import StrOutputParser
from langchain_core.callbacks import StreamingStdOutCallbackHandler

# 프롬프트 템플릿
prompt = ChatPromptTemplate.from_template(
    """<s>A chat between a curious user and an artificial intelligence assistan
t.
The assistant gives helpful, detailed, and polite answers to the user's questi
ons.</s>
<s>Human: {question}</s>
<s>Assistant:"""
)

# GPT4All 로컬 모델 초기화
```

```
llm = GPT4All(
    model=local_path,        # 로컬 모델 경로
    backend="gpu",           # GPU 실행 (또는 "cpu")
    streaming=True,          # 실시간 출력
    callbacks=[StreamingStdOutCallbackHandler()],
)

# 체인 구성
chain = prompt | llm | StrOutputParser()

# 질문 실행
response = chain.invoke({"question": "대한민국의 수도는 어디인가요?"})
```

출력 예시

> 대한민국의 수도는 서울입니다.
>
> 서울은 한반도 북부에 위치한 약 천만 명의 인구를 가진 대도시로,
>
> 한국의 문화·경제·정치의 중심지입니다...

4️⃣ 주요 옵션 정리

옵션	설명
`model`	로컬 `.gguf` / `.bin` 모델 파일 경로
`backend`	`cpu`, `gpu`, `metal` 등 하드웨어 옵션
`streaming`	True일 경우 실시간 응답 스트리밍
`callbacks`	스트리밍 시 출력 제어 핸들러 등록
`n_threads`	CPU 병렬 처리 스레드 수 조정 가능

5️⃣ 장점 요약

항목	설명
완전 오프라인 실행	인터넷 없이 LLM 사용 가능
GPU 가속 지원	빠른 추론 속도 제공
LangChain 통합	체인 기반 프롬프트 연결 가능
다양한 모델 지원	LLaMA, Falcon, Mistral 등 호환
무료 사용	클라우드 API 비용 없음

6️⃣ 폴더 구조 예시

```
project/
│
├── models/
│   └── EEVE-Korean-Instruct-10.8B-v1.0-Q8_0.gguf
│
├── main.py
└── requirements.txt
```

요약

구분	내용
플랫폼	GPT4All (nomic-ai)
실행 방식	로컬 실행 (CPU / GPU 지원)
모델 형식	`.gguf` (LLAMA 계열)
프롬프트 통합	LangChain `GPT4All` 클래스
주요 장점	오프라인 실행, 무료, 간단한 설정

XI. 비디오(Video) 질의 응답 LLM (Gemini)

Gemini 비디오(Video) 질의응답 LLM

개요

Google **Gemini API**는 **비디오 파일을 직접 업로드하여 LLM 기반으로 분석·요약·질의응답**을 수행할 수 있다.

`File API`를 통해 업로드된 영상은 Gemini 모델(`gemini-1.5-flash`)에서 처리되며,

내용 기반 요약, 특정 장면 질의, 대사 인식 등 다양한 질문이 가능하다.

> ⚠️ API Key로 업로드된 파일은 개인 클라우드 프로젝트와 연결되며, 접근 권한이 API Key와 동일하므로 보안에 유의해야 함.

1️⃣ 환경 설정

■ API Key 설정

`.env` 파일에 API Key 저장

```
GOOGLE_API_KEY=<YOUR_API_KEY>
```

■ Python 환경 로드

```
from dotenv import load_dotenv
from langchain_teddynote import logging

load_dotenv()
logging.langsmith("CH04-Gemini-Video")
```

2️⃣ 비디오 업로드

■ 파일 다운로드 (예시)

```
wget "https://www.dropbox.com/scl/fi/ugue14fyo010jgc7wuh4g/teddynote-sample-video.mp4?dl=1" -qO teddynote-sample-video.mp4
```

■ 업로드 코드

```
import google.generativeai as genai

video_file_name = "teddynote-sample-video.mp4"

print("파일을 업로드 중입니다...")
video_file = genai.upload_file(path=video_file_name)
print(f"업로드 완료: {video_file.uri}")
```

3️⃣ 업로드 상태 확인

```
import time

while video_file.state.name == "PROCESSING":
    print("비디오 업로드 및 전처리가 완료될 때까지 잠시만 기다려주세요...")
```

```
    time.sleep(10)
    video_file = genai.get_file(video_file.name)

if video_file.state.name == "FAILED":
    raise ValueError(video_file.state.name)

print(f"비디오 처리가 완료되었습니다! 🎬\nURI: {video_file.uri}")
```

출력 예시

```
비디오 처리가 완료되었습니다!
URI: https://generativelanguage.googleapis.com/v1beta/files/b1d0iktswld
```

4️⃣ 비디오 질의 (요약 질문)

```
prompt = "이 영상에 대해서 짧게 요약해 줄 수 있나요?"

model = genai.GenerativeModel(model_name="models/gemini-1.5-flash")

response = model.generate_content(
    [prompt, video_file],
    request_options={"timeout": 600}
)

print(response.text)
```

출력 예시

> 이 영상은 PDF 파일을 파싱하는 방법에 대한 튜토리얼입니다.
>
> 랭체인과 업스테이지 레이아웃 분석 알고리즘을 사용해
>
> PDF에서 텍스트·테이블·이미지를 추출하는 방법을 설명합니다.

5️⃣ 스트리밍 질의응답

```
prompt = "이 영상에서 Gencon 관련 언급한 부분의 시간을 알려주고, 어떤 내용을 말했는지 알려주세요."
```

```
response = model.generate_content(
    [prompt, video_file],
    request_options={"timeout": 600},
    stream=True
)

for chunk in response:
    print(chunk.text, end="", flush=True)
```

출력 예시

> Gencon에 대한 언급은 0:27부터 나옵니다.
>
> "오늘 또 이벤트가 있습니다. 젠콘, 젠콘 이벤트 무료로 세 분께 드리는..."

6️⃣ 파일 삭제

```
genai.delete_file(video_file.name)
print(f"영상을 삭제했습니다: {video_file.uri}")
```

출력 예시

> 영상을 삭제했습니다:
> https://generativelanguage.googleapis.com/v1beta/files/b1d0iktswld

> ⚠️ 업로드된 파일은 2일 후 자동 삭제되며, 직접 삭제도 가능.

⚙️ 제약사항 요약

항목	설명
파일 크기 제한	2GB 이하
프로젝트 저장 용량	최대 20GB
파일 보관 기간	2일 후 자동 삭제
다운로드 불가	업로드된 파일은 API로 다운로드 불가
보안 주의	API Key로 동일 접근 권한 부여됨

💡 활용 예시

응용	설명
요약 생성	영상 전체 내용 요약
타임스탬프 기반 질의	특정 주제 언급 시점 찾기
내용 기반 질의응답	인물 발언, 강의 핵심 정리
콘텐츠 태깅	장면별 키워드 자동 생성

🔍 모델 요약

항목	값
모델명	`gemini-1.5-flash`
입력형식	텍스트 + 비디오
출력형식	텍스트 (또는 스트리밍)
응답시간	약 10~60초 (파일 크기 의존)
주요 기능	비디오 이해, 장면 요약, 음성 텍스트 인식