CHAPTER 1

LLM(대규모 언어 모델) 기초

인공지능의 새로운 패러다임

LLM이란 무엇인가?

대규모 언어 모델의 정의

LLM은 방대한 양의 텍스트 데이터를 학습하여 인간과 유사한 방식으로 언어를 이해하고 생성할 수 있는 AI 시스템입니다. ChatGPT, Claude, Gemini 등이 대표적인 예입니다.

핵심 작동 방식

- 다음에 올 단어를 예측하는 방식으로 텍스트 생성
- 인터넷의 방대한 데이터로 사전 훈련
- 특정 작업에 맞게 미세 조정 가능

질문 답변

복잡한 질문에 대한 상세한 답변 제공

글쓰기

다양한 스타일의 텍스트 생성

코드 생성

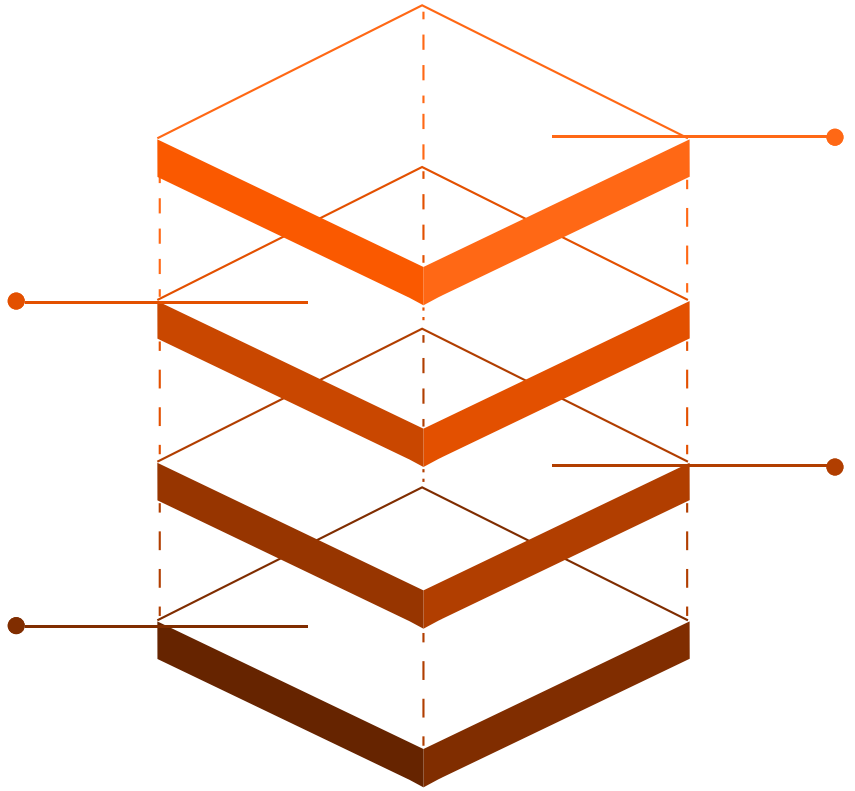
프로그래밍 언어로 작동하는 코드 작성

번역

여러 언어 간 정확한 번역

트랜스포머 아키텍처 쉽게 이해하기

현대의 LLM은 2017년 구글이 발표한 **트랜스포머(Transformer)** 신경망 구조를 기반으로 합니다. 이 혁신적인 아키텍처는 AI 분야에 패러다임의 전환을 가져왔습니다.



디코더
압축된 의미로 새 텍스트 생성

어텐션
단어 간 관계를 동적으로 가중

포지셔널 인코딩
단어 순서 정보 등록

인코더
입력 텍스트 이해 및 의미 압축

GPT 계열은 디코더만 사용하고, BERT는 인코더만 사용하며, 번역 모델은 둘 다 활용합니다.

어텐션 메커니즘: 중요한 것에 집중하기

시험 공부의 비유

트랜스포머의 핵심인 **어텐션(Attention)**은 시험 공부할 때 중요한 부분에 형광펜을 치는 것과 같습니다.

예시: "그녀는 사과를 먹었다. 그것은 맛있었다." 문장에서 "그것"이 무엇인지 이해하려면 "사과"에 더 많은 주의를 기울여야 합니다.

일반적인 읽기 vs 어텐션

일반 읽기: 모든 내용을 같은 집중도로 읽음

→ 비효율적, 중요도 구분 불가

어텐션: 중요한 부분에 형광펜을 치며 읽음

→ 핵심에 집중, 맥락에 따라 동적 판단

토큰과 토큰화

토큰이란?

LLM은 텍스트를 그대로 처리하지 않고 **토큰(Token)**이라는 작은 단위로 쪼개서 처리합니다. 토큰은 단어, 단어의 일부(서브워드), 또는 개별 문자일 수 있습니다.

토큰화 예시

> **원문:** "인공지능이 세상을 바꾸고 있습니다"
>
> **토큰화 결과:** ["인공", "지능", "이", "세상", "을", "바꾸", "고", "있습니다"]

영어

"Hello, how are you?"

→ 약 5-6개 토큰

→ 대략 4글자당 1토큰

한국어

"안녕하세요, 어떻게 지내세요?"

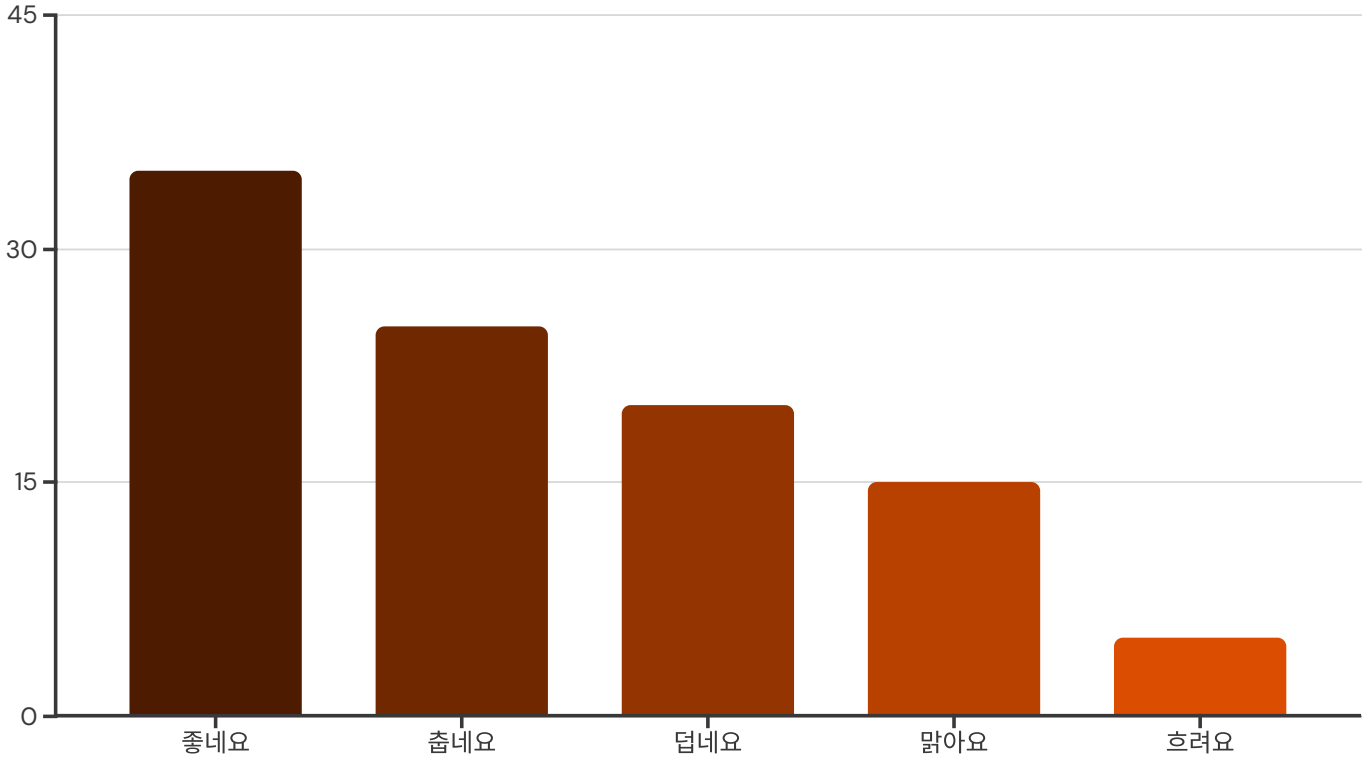
→ 약 10-15개 토큰

→ 대략 1-2글자당 1토큰

한국어는 영어보다 같은 의미를 표현하는 데 더 많은 토큰이 필요하므로 API 비용 계획 시 고려해야 합니다.

다음 단어 예측 원리

LLM의 핵심 작동 원리는 놀라울 정도로 단순합니다: **"지금까지의 텍스트를 보고, 다음에 올 단어를 예측한다"**



확률 분포와 샘플링

입력: "오늘 날씨가 정말"

모델은 확률 분포에서 하나를 선택하고, 이 과정을 반복하여 문장을 완성합니다.

Temperature로 무작위성 조절

- **0:** 항상 가장 확률 높은 단어 (일관된 응답)
- **0.7:** 적당한 무작위성 (균형잡힌 창의성)
- **1.5:** 높은 무작위성 (창의적이지만 예측 불가능)

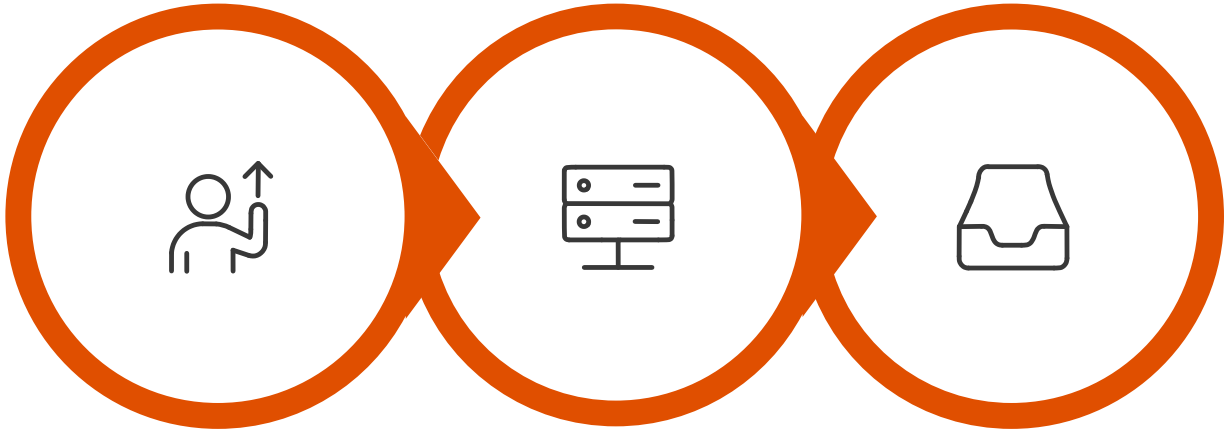
CHAPTER 2

OpenAI API 이해하기

프로그램과 AI가 대화하는 방법

API란?

API(Application Programming Interface)는 서로 다른 프로그램이 소통하는 규칙과 방법입니다. ChatGPT 웹사이트를 거치지 않고 자신의 프로그램에서 직접 GPT를 호출할 수 있게 해줍니다.



이를 통해 자동화된 워크플로우, 맞춤형 애플리케이션, 대규모 처리가 가능합니다.

Chat Completion API 구조

요청(Request) 구조

```
{
  "model": "gpt-4.1-mini",
  "messages": [
    {"role": "system", "content": ...},
    {"role": "user", "content": ...}
  ],
  "temperature": 0.7,
  "max_tokens": 1000
}
```

JSON 형식으로 모델, 메시지, 매개변수를 지정합니다.

응답(Response) 구조

```
{
  "id": "chatcmpl-...",
  "model": "gpt-4.1-mini",
  "choices": [{
    "message": {
      "role": "assistant",
      "content": "응답 텍스트..."
    }
  }],
  "usage": {
    "total_tokens": 150
  }
}
```

AI의 응답과 토큰 사용량을 포함합니다.

메시지 역할의 이해

Chat Completion API에서 메시지는 세 가지 역할을 가집니다. 각 역할은 대화의 맥락과 AI의 행동 방식을 결정합니다.

system

AI의 성격과 규칙 정의

"당신은 친절한 상담원입니다"

user

사용자의 질문

"파이썬 리스트 정렬 방법은?"

assistant

AI의 응답

"sort() 메서드를 사용하세요"

이전 user/assistant 메시지를 포함하면 대화 맥락이 유지되어 자연스러운 멀티턴 대화가 가능합니다.

핵심 매개변수: Temperature

Temperature는 LLM의 창의성을 조절하는 가장 중요한 매개변수입니다. 0에서 2 사이의 값으로 설정합니다.

기타 핵심 매개변수

top_p

단어 선택 범위를 확률 기준으로 제한 (nucleus sampling)

- 0.1: 상위 10% 확률 단어만 고려
- 0.9: 상위 90% 확률 단어 고려 (권장)

max_tokens

생성되는 응답의 최대 토큰 수 제한

- 100: 짧은 답변 (50-75 단어)
- 500: 중간 길이
- 2000: 긴 답변

frequency_penalty

이미 많이 나온 단어의 재등장 억제 (-2 ~ 2)

양수: 반복 방지 | 음수: 반복 허용

presence_penalty

새로운 주제로의 전환 유도 (-2 ~ 2)

양수: 주제 전환 | 음수: 현재 주제 집중

토큰 수 추정과 비용 최적화

토큰 수 추정 규칙

영어

1 토큰 ≈ 4글자 ≈ 0.75단어

"Hello, how are you?" ≈ 5-6 토큰

한국어

1 토큰 ≈ 1-2글자

"안녕하세요" ≈ 3-5 토큰

정확한 계산은 OpenAI의 `tiktoken` 라이브러리 또는 Tokenizer 웹 도구를 활용하세요.

비용 최적화 전략

1. **적절한 모델 선택:** 간단한 작업에는 gpt-4.1-mini, gpt-4.1-nano 등 저렴한 모델 사용
2. **프롬프트 최적화:** 불필요한 지시문 제거, 핵심만 전달
3. **max_tokens 제한:** 필요한 만큼만 응답 받기
4. **캐싱 활용:** 동일 요청 결과를 저장해두고 재사용

고급 기능 소개

멀티모달

최신 GPT-4 계열 모델은 텍스트뿐 아니라 이미지와 오디오도 처리할 수 있습니다. URL 또는 Base64로 이미지를 전달하고, 텍스트 응답을 음성으로 변환할 수 있습니다.

Function Calling

LLM이 외부 함수를 호출할 수 있게 합니다. 예: "서울 날씨 알려줘" → LLM이 날씨 API 호출 판단 → 결과를 자연어로 변환하여 응답

Streaming

응답을 한 번에 받지 않고 토큰 단위로 실시간 수신합니다. ChatGPT처럼 글자가 순차적으로 나타나 사용자 체감 속도가 향상됩니다.

CHAPTER 3

LangChain 프레임워크 소개

복잡한 LLM 애플리케이션을 쉽게

왜 LangChain이 필요한가?

API 직접 호출의 한계

- 반복적인 코드 작성 (API 클라이언트 설정, 에러 처리 등)
- 복잡한 워크플로우 구현의 어려움
- 모델 교체 시 코드 전면 수정 필요
- 각 제공자마다 다른 API 형식

프레임워크의 장점

모듈성: 레고 블록처럼 컴포넌트 조합

통일된 인터페이스: OpenAI, Anthropic, Google 등을 같은 코드로 사용

생산성: 검증된 패턴과 도구 제공

LangChain 아키텍처 개요

LangChain은 다양한 컴포넌트로 구성된 모듈식 프레임워크입니다. 각 컴포넌트는 독립적으로 작동하며 필요에 따라 조합할 수 있습니다.

언어 처리

Models, Prompts, Output Parsers

문서 처리

Document Loaders, Text Splitters, Embeddings

저장/검색

Vector Stores, Retrievers

실행

Chains, Agents

RAG 파이프라인 예시

Document Loader → Text Splitter → Embeddings → Vector Store

User Query → Retriever → Prompt + Context → LLM → Output Parser

필요한 컴포넌트만 선택하여 맞춤형 파이프라인을 구성할 수 있습니다.

모델 래퍼의 통일된 인터페이스

LangChain은 각 AI 제공자의 API를 래퍼로 감싸서 통일된 방식으로 사용할 수 있게 합니다.

OpenAI 모델

```
from langchain_openai import ChatOpenAI

model = ChatOpenAI(
  model="gpt-4.1-mini"
)

response = model.invoke(
  "안녕하세요!"
)
```

Google 모델 (같은 방식!)

```
from langchain_google_genai
  import ChatGoogleGenerativeAI

model = ChatGoogleGenerativeAI(
  model="gemini-2.5-flash-lite"
)

response = model.invoke(
  "안녕하세요!"
)
```

모델만 교체하면 나머지 코드는 그대로 사용할 수 있어 비교 테스트와 유지보수가 간편합니다.

메시지 타입 심화

LangChain은 OpenAI API의 메시지 역할을 클래스로 표현합니다.

1

1차 대화

SystemMessage("당신은 비서입니다")

HumanMessage("제 이름은 철수입니다")

→ AI: "안녕하세요, 철수님!"

2

2차 대화 (기록 유지)

이전 SystemMessage, HumanMessage 포함

AIMessage("안녕하세요, 철수님!") ← 추가

HumanMessage("제 이름이 뭐였죠?")

→ AI: "철수님이시죠!" ← 기억함

이전 대화를 메시지 리스트에 포함시키면 AI가 맥락을 기억합니다.

CHAPTER 4

프롬프트 엔지니어링 기초

AI에게 보내는 효과적인 지시문

프롬프트란?

좋은 프롬프트의 조건

프롬프트는 AI에게 원하는 작업을 수행하도록 하는 지시문입니다. 좋은 프롬프트는 명확하고 구체적이며 AI가 이해하기 쉬운 형태로 작성됩니다.

명확성

목적이 분명함

구체성

세부 사항 포함

맥락

충분한 배경 정보

나쁜 프롬프트 vs 좋은 프롬프트

> ~~나쁜 예:~~ "글 써줘"
>
> → 너무 모호함, 어떤 글? 주제는? 길이는?

> **좋은 예:** "초등학생을 대상으로 지구 온난화의 원인과 영향을 3문단, 총 200자 내외로 설명하는 글을 작성해주세요."
>
> → 대상, 주제, 형식, 길이가 명확함

프롬프트 템플릿의 필요성

하드코딩 방식 (비추천)

```
prompt1 = "파이썬에 대해 설명해주세요"
prompt2 = "자바스크립트에 대해 설명해주세요"
prompt3 = "러스트에 대해 설명해주세요"

# → 같은 패턴을 반복 작성
# → 유지보수 어려움
```

템플릿 사용 (추천)

```
template = "{topic}에 대해 설명해주세요"

prompt1 = template.format(
 topic="파이썬"
)
prompt2 = template.format(
 topic="자바스크립트"
)

# → 변수만 바꿔서 재사용
# → 한 곳만 수정하면 전체 적용
```

템플릿 종류와 선택 가이드

1

PromptTemplate

용도: 단순 텍스트 생성

템플릿: "{topic}에 대해 {length}자로 설명"

출력: "인공지능에 대해 200자로 설명"

시나리오: 단순 질문-답변, 텍스트 생성

2

ChatPromptTemplate

용도: 대화형 AI 구성

system: "당신은 {role}입니다"

user: "{question}"

시나리오: 페르소나 설정, 역할 기반 응답

3

MessagesPlaceholder

용도: 동적 메시지 삽입

대화 기록을 템플릿에 삽입

chat_history 변수에 이전 대화 전달

시나리오: 멀티턴 대화, 히스토리 유지

변수 바인딩과 동적 프롬프트

템플릿의 변수에 실제 값을 넣는 것을 **바인딩(Binding)**이라고 합니다.

예시 코드

```
template = "{product}의 장점을 {count}가지 알려주세요"

variables = {
  "product": "아이폰",
  "count": "3"
}

prompt = template.invoke(variables)
```

결과

> "아이폰의 장점을 3가지 알려주세요"

변수를 바꾸면 다양한 제품, 다양한 개수로 프롬프트를 동적으로 생성할 수 있습니다.

CHAPTER 5

출력 파서와 구조화된 응답

프로그래밍 가능한 데이터로

왜 출력을 파싱해야 하는가?

자유 텍스트의 한계

질문: "아이폰 15 프로의 정보를 알려줘"

LLM 응답: "아이폰 15 프로는 애플에서 2023년에 출시한 스마트폰입니다. 가격은 약 150만원이고..."

문제: 이 텍스트에서 가격만 추출하려면? 출시년도를 변수로 사용하려면? 다른 제품과 비교 표를 만들려면?

구조화된 출력

```
{
  "brand": "애플",
  "model": "아이폰 15 프로",
  "year": 2023,
  "price": 1500000,
  "storage": "256GB"
}
```

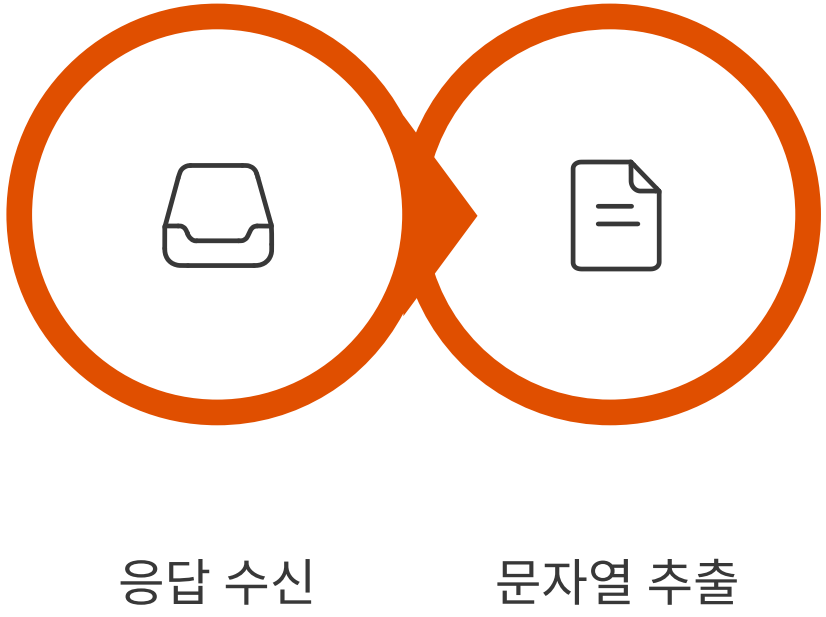
→ 가격 추출: data["price"]

→ 년도 사용: data["year"]

→ 비교 표: 데이터프레임으로 쉽게 변환

StrOutputParser

가장 기본적인 파서로, AI 응답 객체에서 텍스트만 추출합니다.



사용 시나리오: 응답 텍스트만 필요할 때, 체인의 마지막 단계에서 간단하게 문자열을 얻고 싶을 때

구조화된 출력 (Structured Output)

1. Pydantic 모델 정의

```
class ProductInfo(BaseModel):
  """상품 정보를 담는 구조"""
  brand: str = Field(
    description="제조사"
  )
  model: str = Field(
    description="모델명"
  )
  price: int = Field(
    description="가격(원)"
  )
  category: str = Field(
    description="카테고리"
  )
```

2. with_structured_output 사용

```
structured_model = llm.with_structured_output(
  ProductInfo
)

result = chain.invoke({
  "product_text": 제품_설명_텍스트
})

# 결과 사용
print(result.brand)    # 제조사
print(result.model)    # 모델명
print(result.price)    # 가격
```

LLM이 자동으로 정의된 구조에 맞는 데이터를 반환합니다.

CHAPTER 6

LCEL(LangChain Expression Language)

선언적 체인 구성 언어

LCEL이란?

LCEL은 LangChain 컴포넌트들을 파이프 연산자(|)로 연결하여 워크플로우를 구성하는 표현 언어입니다.

LCEL 없이 (명령형)

```
prompt_result = prompt.invoke({
  "topic": "AI"
})

llm_result = llm.invoke(
  prompt_result
)

final_result = parser.invoke(
  llm_result
)
```

→ 단계마다 중간 변수 필요

→ 코드가 길어지고 복잡함

LCEL 사용 (선언형)

```
chain = prompt | llm | parser

final_result = chain.invoke({
  "topic": "AI"
})
```

→ 간결하고 읽기 쉬움

→ 데이터 흐름이 명확함

→ 유닉스 파이프와 유사한 직관적 구조

파이프 연산자(|) 완전 정복

LCEL의 | 연산자는 유닉스 셸의 파이프와 같은 개념으로, 각 단계의 출력이 다음 단계의 입력이 됩니다.

체인 구성 패턴

01

Prompt | LLM

기본 패턴: 프롬프트 생성 후 LLM 호출

02

Prompt | LLM | Parser

완전한 패턴: 출력까지 처리

Runnable 인터페이스 종류

LCEL의 모든 컴포넌트는 **Runnable** 인터페이스를 구현합니다.

RunnableSequence

순차 실행: 여러 Runnable을 순서대로 실행

Step 1 → Step 2 → Step 3

각 단계의 출력이 다음 입력

RunnableParallel

병렬 실행: 동일 입력을 여러 Runnable에 동시 전달

같은 질문을 여러 모델에 동시 전송

결과를 딕셔너리로 수집

RunnablePassthrough

입력 그대로 전달: 원본 입력 유지

RunnableParallel과 함께 사용

원본과 가공 결과 동시 유지

RunnableLambda

커스텀 함수: 일반 함수를 Runnable로 변환

전처리/후처리 로직 삽입

중간 단계 커스텀 처리

실행 메서드 비교

메서드	설명
invoke	단일 입력 처리 (동기)
batch	여러 입력 일괄 처리 (동기)
stream	스트리밍 출력 (동기)
ainvoke	단일 입력 처리 (비동기)
astream	스트리밍 출력 (비동기)

선택 가이드

단일 요청

invoke 사용

여러 요청 한 번에

batch 사용

실시간 출력

stream 사용

웹 서버/비동기

ainvoke, astream 사용