# LangChin AI Agen 만들기1

오산개발자모임

# 1. LangChin이란?

**LangChain**은 대형 언어 모델(Large Language Model, LLM)을 활용한 애플리케이션 개발을 위한 **오픈소스 프레임워크** 입니다.

### 🎯 핵심 개념

- **체인(Chain)**: 여러 컴포넌트를 연결하여 복잡한 작업을 수행
- **모듈화**: 재사용 가능한 컴포넌트들의 조합
- **추상화**: LLM과의 상호작용을 간단하게 만드는 고수준 인터페이스

### ✨ 주요 특징

- **다양한 LLM 지원**: OpenAI, Google PaLM, Anthropic Claude, 로컬 모델 등
- **풍부한 통합**: 벡터 데이터베이스, 메모리, 도구 등과의 원활한 연동
- **유연한 구조**: 필요에 따라 컴포넌트를 선택하고 조합
- python, javascript SDK 제공, MIT license

### 🔮 두 레이어로 설계된 **Composable** 구조

- **Components Layer** : Prompt, LLM Wrapper, Chain, Tool, Memory, VectorStore 등 저수준 블럭
- **Use-cases Layer** : ChatBot, 데이터어거스팅 , RAG, Agent 등 고수준 템플릿

# 2. 왜 LangChain을 사용하는가?

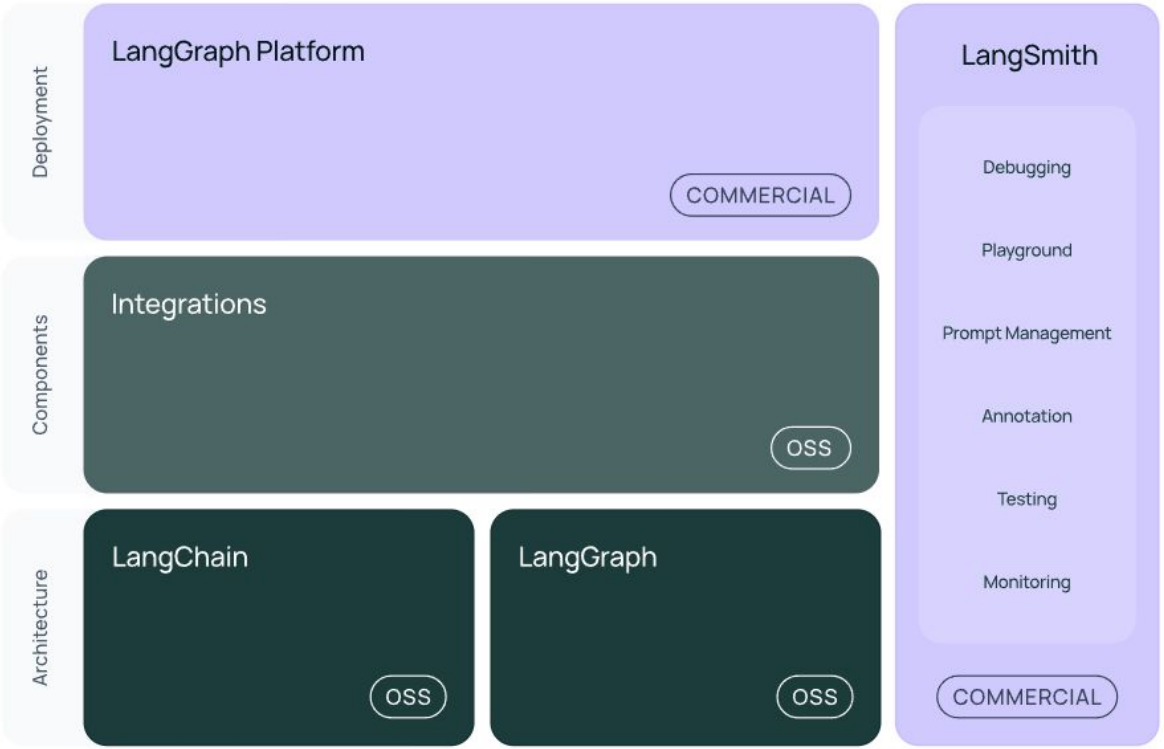
```python
# LangChain 없이
response = openai.Completion.create(
    engine="text-davinci-003",
    prompt="질문: 파리의 인구는? 답변:",
    max_tokens=150
)

# LangChain 사용
from langchain.llms import OpenAI
llm = OpenAI()
response = llm("파리의 인구는?")
```

### 💡 주요 장점

- **빠른 프로토타이핑** : 몇 줄의 코드로 복잡한 AI 애플리케이션 구현
- **표준화된 인터페이스** : 다양한 LLM을 일관된 방식으로 사용
- **확장성** : 필요에 따라 기능을 추가하고 확장 가능
- **커뮤니티 지원** : 활발한 개발자 커뮤니티와 풍부한 문서

# 3. LangChain Structure




https://python.langchain.com

# 4. LangChain Python API Reference

## Base packages #

| Core | Langchain | Text Splitters |
| --- | --- | --- |
| langchain-core: 0.3.59 | langchain: 0.3.25 | langchain-text-splitters: 0.3.8 |

| Community | Experimental |
| --- | --- |
| langchain-community: 0.3.24 | langchain-experimental: 0.3.5rc1 |

## Integrations #

| OpenAI | Anthropic | Google VertexAI |
| --- | --- | --- |
| langchain-openai 0.3.16 | langchain-anthropic 0.3.13 | langchain-google-vertexai 2.0.23 |

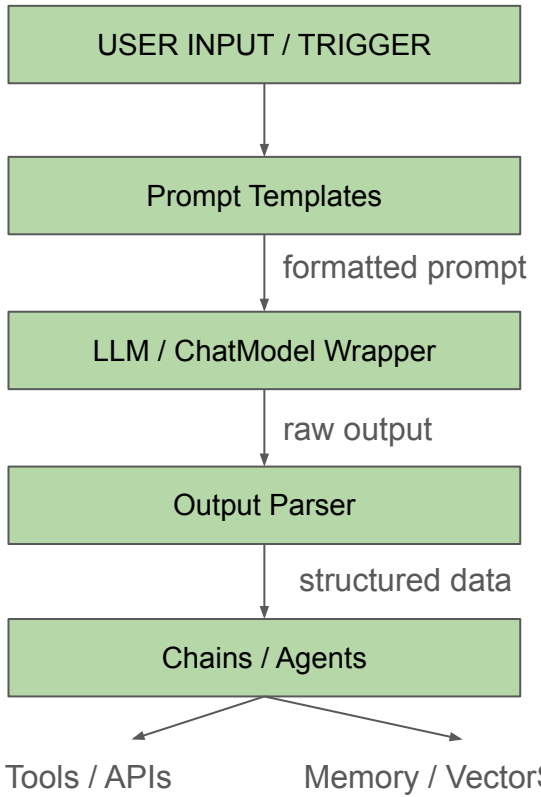
| AWS | Huggingface | MistralAI |
| --- | --- | --- |
| langchain-aws 0.2.23 | langchain-huggingface 0.2.0 | langchain-mistralai 0.2.10 |

https://python.langchain.com/api_reference/

# 5. LangChain 핵심 개념

| 개념 | 설명 | 주요 API |
| --- | --- | --- |
| **Prompt Template** | LLM 입력을 위한 파라미터화 템플릿 | PromptTemplate, ChatPromptTemplate |
| **LLM & ChatModel** | OpenAI, Azure, Ollama, Claude 등 래퍼 | OpenAI, ChatOpenAI, LlamaCpp |
| **Output Parser** | LLM 출력을 구조화(예: JSON) | StrOutputParser, PydanticOutputParser |
| **Chains** | 여러 단계 호출을 직렬/분기로 연결 | LLMChain, SequentialChain, RouterChain |
| **Agents** | 도구 선택·실행을 스스로 계획하는 실행기 | AgentExecutor, ConversationalAgent |
| **Tools** | 외부 기능 래퍼(검색, DB, API) | Tool, PythonREPLTool, SerpAPIWrapper |
| **Memory** | 이전 대화를 저장·재사용 | ConversationBufferMemory, SummaryMemory |
| **Retrievers & VectorStores** | RAG 구현용 벡터 DB 인터페이스 | FAISS, Chroma, SupabaseVectorStore |
| **Callbacks & Tracing** | 실행 로그·토큰·latency 추적 | CallbackHandler, LangSmith UI |

# 6.LangChain의 구성 요소 아키텍처




◂ 프롬프트 엔지니어 레이어

◂ 모델 엑세스 레이어

◂ 구조화 레이어

◂ 오케스트레이션 레이어

◂ 확장 컴포넌트

# 7. Model

LLM과의 인터페이스를 제공하는 컴포넌트

**주요 타입:**

- **LLMs**: 텍스트 생성 모델 (GPT-3, PaLM 등)
- **Chat Models**: 대화형 모델 (GPT-4, Claude 등)
- **Embeddings**: 텍스트를 벡터로 변환하는 모델

```python
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.embeddings import OpenAIEmbeddings

# LLM 초기화
llm = OpenAI(temperature=0.7)

# Chat Model 초기화
chat_model = ChatOpenAI(model_name="gpt-4")

# Embeddings 초기화
embeddings = OpenAIEmbeddings()
```

# 8. Prompts

모델에 전달할 입력을 관리하는 컴포넌트

**주요 기능:**

- **템플릿화**: 동적으로 프롬프트 생성
- **검증**: 입력 데이터 유효성 검사
- **최적화**: 프롬프트 성능 향상

```python
from langchain.prompts import PromptTemplate

# 프롬프트 템플릿 정의
template = """
당신은 도움이 되는 AI 어시스턴트입니다.
질문: {question}
답변: """

prompt = PromptTemplate(
    input_variables=["question"],
    template=template
)

# 프롬프트 생성
formatted_prompt = prompt.format(question="파이썬이란 무엇인가요?")
```

# 9. Chains

여러 컴포넌트를 연결하여 복잡한 워크플로우를 구성

**주요 타입:**

- **LLMChain**: 프롬프트 + LLM의 기본 조합
- **SequentialChain**: 순차적으로 체인 실행
- **MapReduceChain**: 병렬 처리 후 결과 통합

```python
from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

llm = OpenAI()
prompt = PromptTemplate(
    input_variables=["topic"],
    template="다음 주제에 대한 설명을 작성해주세요: {topic}"
)

chain = LLMChain(llm=llm, prompt=prompt)
result = chain.run("인공지능")
```

# 10. Memory

대화의 문맥과 이전 상호작용을 기억

**주요 타입:**

- **ConversationBufferMemory**: 전체 대화 내용 저장
- **ConversationSummaryMemory**: 대화 요약 저장
- **ConversationBufferWindowMemory**: 최근 N개 메시지만 저장

```python
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain

memory = ConversationBufferMemory()
conversation = ConversationChain(
    llm=llm,
    memory=memory,
    verbose=True
)

conversation.predict(input="안녕하세요!")
conversation.predict(input="제 이름은 김철수입니다.")
conversation.predict(input="제 이름이 뭐라고 했나요?")
```

# 11. Agents

동적으로 행동을 결정하고 도구를 사용하는 컴포넌트

**주요 특징:**

- **추론**: 어떤 행동을 취할지 결정
- **도구 사용**: 외부 API나 도구 활용
- **반복 처리**: 목표 달성까지 반복 실행

```python
from langchain.agents import initialize_agent, AgentType
from langchain.tools import DuckDuckGoSearchRun

# 도구 초기화
search = DuckDuckGoSearchRun()

# 에이전트 초기화
agent = initialize_agent(
    tools=[search],
    llm=llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)

# 에이전트 실행
result = agent.run("2024년 한국의 인구는 얼마인가요?")
```

# 12. Tools

외부 시스템과의 연동을 위한 인터페이스

**주요 카테고리 :**

- **검색 도구**: Google Search, Wikipedia
- **계산 도구**: Calculator, Wolfram Alpha
- **API 도구**: REST API, 데이터베이스 연결

```python
from langchain.tools import DuckDuckGoSearchRun
from langchain.agents import initialize_agent

# 검색 도구 초기화
search = DuckDuckGoSearchRun()

# 에이전트 생성
search_agent = initialize_agent(
    tools=[search],
    llm=llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)

# 검색 실행
result = search_agent.run("2024년 대한민국 GDP는?")
```

# 13. Retrievers (검색기)

문서에서 관련 정보를 검색하는 컴포넌트

**주요 타입:**

- **VectorStoreRetriever**: 벡터 유사도 기반 검색
- **BM25Retriever**: 키워드 기반 검색
- **HybridRetriever**: 여러 방법 조합

```python
from langchain.retrievers import BM25Retriever

# 문서 텍스트 리스트
texts = [
    "LangChain은 LLM 애플리케이션 개발을 위한 프레임워크입니다.",
    "Python은 머신러닝과 딥러닝에 활용됩니다.",
    "OpenAI GPT-4는 강력한 대화형 AI 모델입니다.",
    "벡터 데이터베이스는 임베딩 저장에 사용됩니다."
]

# BM25 Retriever 생성
bm25_retriever = BM25Retriever.from_texts(texts)
bm25_retriever.k = 2  # 상위 2개 결과 반환

# 검색 실행
query = "LangChain 프레임워크"
results = bm25_retriever.get_relevant_documents(query)

for doc in results:
    print(f"결과: {doc.page_content}\n")
```

# 14. Document Loaders

다양한 소스에서 문서를 로드하는 컴포넌트

**지원 형식:**

- PDF, DOCX, TXT
- CSV, JSON, XML
- 웹페이지, GitHub, Notion

```python
from langchain.document_loaders import PyPDFLoader

# PDF 파일 로드
loader = PyPDFLoader("document.pdf")
pages = loader.load_and_split()

print(f"총 페이지 수: {len(pages)}")

# 각 페이지별로 처리
for i, page in enumerate(pages[:3]):
    print(f"페이지 {i+1}:")
    print(f"  내용 미리보기: {page.page_content[:100]}...")
    print(f"  메타데이터: {page.metadata}\n")
```

# 15. Hello LangChain

📦 설치

```bash
# LangChain 설치
pip install langchain

# OpenAI 사용을 위한 추가 패키지
pip install openai

# 기타 유용한 패키지들
pip install python-dotenv
pip install tiktoken
```

🔑 API 키 설정

```python
import os
from dotenv import load_dotenv

# .env 파일 로드
load_dotenv()

# 환경 변수 설정
os.environ["OPENAI_API_KEY"] = "your-api-key-here"
```

# 15. Hello LangChain

```python
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

# 1. LLM 초기화
llm = OpenAI(temperature=0.9)

# 2. 프롬프트 템플릿 생성
prompt = PromptTemplate(
    input_variables=["product"],
    template="다음 제품에 대한 마케팅 문구를 작성해주세요: {product}"
)

# 3. 체인 생성
chain = LLMChain(llm=llm, prompt=prompt)

# 4. 실행
result = chain.run("스마트워치")
print(result)
```

“실패는 하나의 옵션입니다. 만약 실패를 겪지 않았다면 당신은 충분히 혁신적이지 않았다는 증거 입니다.”

– 일런머스크