
- 출처: LangChain 공식 문서 또는 해당 교재명
- 원본 URL: https://smith.langchain.com/hub/teddynote/summary-stuff-documents

---

LCEL(대화내용 기억하기) : 메모리 추가

- 임의의 체인에 메모리 추가하기 (*수동으로 연결해야 함*)

```
# 환경변수 처리 및 클라이언트 생성
from langsmith import Client
from langchain.prompts import PromptTemplate
from langchain.prompts import ChatPromptTemplate

from dotenv import load_dotenv

import os
import json

# 클라이언트 생성
api_key = os.getenv("LANGSMITH_API_KEY")
client = Client(api_key=api_key)
```

```
# LangSmith 추적 설정하기 (https:smith.langchin.com)
# LangSmith 추적을 위한 라이브러리 임포트
from langsmith import traceable

# LangSmith 환경 변수 확인

print("\n--- LangSmith 환경 변수 확인 ---")
langchain_tracing_v2 = os.getenv('LANGCHAIN_TRACING_V2')
langchain_project = os.getenv('LANGCHAIN_PROJECT')
langchain_api_key_status = "설정됨" if os.getenv('LANGCHAIN_A
org = "설정됨" if os.getenv('LANGCHAIN_ORGANIZATION') else "

if langchain_tracing_v2 == "true" and os.getenv('LANGCHAIN_
    print(f"✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='
    print(f"✅ LangSmith 프로젝트: '{langchain_project}'")
    print(f"✅ LangSmith API Key: {langchain_api_key_status
    print("  -> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요
else:
    print("❌ LangSmith 추적이 완전히 활성화되지 않았습니다. 다음을 확
    if langchain_tracing_v2 != "true":
        print(f"  - LANGCHAIN_TRACING_V2가 'true'로 설정되어 있
    if not os.getenv('LANGCHAIN_API_KEY'):
        print("  - LANGCHAIN_API_KEY가 설정되어 있지 않습니다.")
    if not langchain_project:
        print("  - LANGCHAIN_PROJECT가 설정되어 있지 않습니다.")
```

"@traceable" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

- 셀 출력

```
--- LangSmith 환경 변수 확인 ---
✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='true')
✅ LangSmith 프로젝트: 'LangChain-prantice'
✅ LangSmith API Key: 설정됨
-> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.
```

```
import os
from dotenv import load_dotenv
import openai

from langchain_openai import ChatOpenAI

# .env 파일에서 환경변수 불러오기
load_dotenv()
```

```
# 환경변수에서 API 키 가져오기
api_key = os.getenv("OPENAI_API_KEY")

# OpenAI API 키 설정
openai.api_key = api_key

# OpenAI를 불러오기
# ✅ 디버깅 함수: API 키가 잘 불러와졌는지 확인
def debug_api_key():
    if api_key is None:
        print("❌ API 키를 불러오지 못했습니다. .env 파일과 변수명을 확인하세요.")
    elif api_key.startswith("sk-") and len(api_key) > 20:
        print("✅ API 키를 성공적으로 불러왔습니다.")
    else:
        print("⚠️ API 키 형식이 올바르지 않은 것 같습니다. 값을 확인하세요.")

# 디버깅 함수 실행
debug_api_key()
```

- 셀 출력

  ✅ API 키를 성공적으로 불러왔습니다.

```
# LLM 생성
llm = ChatOpenAI(
    temperature=0,
    openai_api_key=api_key,
    model="gpt-4o-mini",
    )
```

- **ChatOpenAI 모델 초기화, 대화형 프롬프트 생성하기**

```
from operator import itemgetter
from langchain.memory import ConversationBufferMemory
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.runnables import RunnableLambda, RunnablePassthrough
from langchain_openai import ChatOpenAI

# ChatOpenAI 모델 초기화
model = ChatOpenAI(
    temperature=0,                                          # 반드시 매개변수를 지정해줘야 오류가 발생하지 않음
    openai_api_key=api_key,
    model="gpt-4o-mini",
)

# 대화형 프롬프트 생성
# 프롬프트  = 시스템 메시지, 이전 대화 내역, 사용자 입력 포함
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful chatbot"),
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)
```

ConversationBufferMemory 생성

- 대화내용을 저장할 메모리 = **ConversationBufferMemory** 생성
- **return_messages 매개변수 = True**
  - 생성된 인스턴스가 메시지를 반환하도록 설정함
- **memory_key 설정** → 이후 Chain의 prompt 안에 대입될 key

```
# 대화 버퍼 메모리를 생성 = 메시지 반환 기능을 활성화

memory = ConversationBufferMemory(
    return_messages=True,
    memory_key="chat_history"
    )
```

- 교재에 안내된 셀 출력

```
{'chat_history': []}
```

- 셀 출력 : **경고 메시지 및 참고 가이드 안내**

```
/var/folders/h3/l7wnkv352kqftv0t8ctl2ld40000gn/T/ipykernel_72156/187112523.py:3: LangChainDeprecationWarning: Plea
memory = ConversationBufferMemory(
```

- 셀 출력 해석
  - 파일 경로와 줄 번호

```
/var/folders/h3/.../ipykernel_72156/187112523.py:3:
```

* * 파이썬이 실행중인 스크립트 (or 노트북 내부의 임시파일) 위치 * `:3:` - 그 중 3번째 줄에서 해당 메시지가 발생했다는 의미

- `LangChainDeprecationWarning`

```
LangChainDeprecationWarning:
```

* * **사용 중단 경고 `DeprecationWarning`** = 이 기능은 곧 사라질 예정이니 다른 방법으로 바꿔서 사용하라는 의미 * 당장 코드가 멈추지 않고 실행은 되지만, 이후 버전에서는 지원되지 않음

- 권장 안내 문구

```
Please see the migration guide at: https://python.langchain.com/docs/versions/migrating_memory/
```

* * **migration guide** * 마이그레이션 가이드 주소 안내

RunnablePassthrough.assign

- **`chat_history` 변수에 `memory.load_memory_variables` 함수의 결과를 할당 → `chat_history` 키에 해당하는 값 추출**

```
# runnablepassthrough.assign 사용하기
runnable = RunnablePassthrough.assign(
    chat_history=RunnableLambda(memory.load_memory_variables)
    | itemgetter("chat_history")                                    # memory_key 와 동일하게 입력하기
)

runnable.invoke({"input": "hi"})
```

- 교재 안내되어있는 셀 출력 결과값

```
{'input': 'hi', 'chat_history': {'chat_history': []}}
```

- 셀 출력

```
{'input': 'hi', 'chat_history': []}
```

```
runnable.invoke({"input": "hi"})
```

- 셀 출력

```
{'input': 'hi', 'chat_history': []}
```

```
# 프롬프트 입력하기
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful chatbot"),
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)
```

`runnable`에 첫 번째 대화 시작하기

- **`input`**: 사용자 입력 대화 전달
- **`chat_history`**: 대화 기록 전달

```
runnable.invoke({"input": "hi!"})
```

- 셀 출력

```
{'input': 'hi', 'chat_history': []}
```

```
# 체인 생성

chain = runnable | prompt | model
```

- **첫 번째 대화 진행**

```
# chain 객체의 invoke 메서드 사용 → 입력에 대한 응답 생성하기
response = chain.invoke({"input": "만나서 반갑습니다. 제 이름은 앨리스입니다."})
print(response.content)                                  # 생성된 응답 출력해보기
```

- 셀 출력 (2.3s)

```
안녕하세요, 앨리스! 만나서 반갑습니다. 어떻게 도와드릴까요?
```

- **`memory.save_context` 함수 = 입력 데이터(`inputs`)와 응답 내용(`response.content`)을 메모리에 저장**
- AI 모델의 학습 과정에서 현재 상태를 기록하거나, 사용자의 요청과 시스템의 응답을 추적하는 데 사용

```
# 입력된 데이터와 응답 내용을 메모리에 저장하긱
memory.save_context(
```

```
    {"human": "만나서 반갑습니다. 제 이름은 앨리스입니다."}, {"ai": response.content}
)

# 저장된 대화기록 출력하기
memory.load_memory_variables({})
```

- 셀 출력

```
{'chat_history': [HumanMessage(content='만나서 반갑습니다. 제 이름은 앨리스입니다.', additional_kwargs={}, response_metadat
AIMessage(content='안녕하세요, 앨리스! 만나서 반갑습니다. 어떻게 도와드릴까요?', additional_kwargs={}, response_metadata={})]}
```

- **추가 질문해보기** - 이름 기억 여부

```
# 이름을 기억하고 있는지 추가 질문
response = chain.invoke({"input": "제 이름이 무엇이었는지 기억하세요?"})

# 답변 출력해보기
print(response.content)
```

- 셀 출력 (0.9s)

```
네, 당신의 이름은 앨리스입니다. 다른 질문이나 이야기하고 싶은 것이 있나요?
```

커스텀 ConverstionChain 구현 예시

```
from operator import itemgetter
from langchain.memory import ConversationBufferMemory, ConversationSummaryMemory
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.runnables import RunnableLambda, RunnablePassthrough, Runnable
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(
    model_name="gpt-4o-mini",
    temperature=0)

# 대화형 프롬프트 생성하기
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful chatbot"),
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)

# 대화 버퍼 메모리를 생성하고, 메시지 반환 기능을 활성화하기
memory = ConversationBufferMemory(return_messages=True, memory_key="chat_history")


class MyConversationChain(Runnable):
    # 사용자 정의 대화 체인 클래스
    # Runnable 인터페이스를 구현하여, 체인을 통한 일련의 처리 과정 정의합

    def __init__(self, llm, prompt, memory, input_key="input"):
        # llm: 언어 모델 객체를 전달받음
        # prompt: 프롬프트 템플릿 또는 Runnable 형태의 프롬프트 컴포넌트
        # memory: 대화 기록을 저장·불러오는 메모리 객체
        # input_key: 입력 딕셔너리에서 사용자 질문을 찾기 위한 키 이름

        self.prompt = prompt
        self.memory = memory
        self.input_key = input_key
```

```
        # 체인 구성: 메모리 로드 → 프롬프트 → 언어 모델 → 문자열 파싱
        self.chain = (
            # RunnablePassthrough.assign를 통해 초기 입력에 chat_history 필드 추가
            RunnablePassthrough.assign(
                chat_history=(
                    # memory.load_memory_variables 메서드를 호출해 이전 대화 기록 가져오기
                    RunnableLambda(self.memory.load_memory_variables)
                    # 가져온 변수 중 memory.memory_key(기본값 "chat_history")에 해당하는 값을 꺼내오기
                    | itemgetter(memory.memory_key)
                )
            )
            # 이어서 프롬프트 실행
            | prompt
            # 프롬프트 결과를 언어 모델(llm)에 전달해 응답 생성
            | llm
            # 모델 출력(딕셔너리나 토큰 리스트 등)을 문자열로 변환해 최종 텍스트만 꺼내기
            | StrOutputParser()
        )

    def invoke(self, query, configs=None, **kwargs):
        # query: 사용자가 입력한 질문(스트링)입니다.
        # configs, **kwargs: 추가 구성 옵션(필요 시 사용).

        # 체인 실행: {input_key: query} 형태로 딕셔너리를 넘겨줍니다.
        answer = self.chain.invoke({self.input_key: query})

        # 실행 후, 메모리에 대화 내용을 저장하기
        self.memory.save_context(
            inputs={"human": query},            # inputs: 사용자 발화("human" 키) → query
            outputs={"ai": answer}              # outputs: AI 응답("ai" 키) → answer
        )

        # 최종 생성된 답변 문자열 반환
        return answer


# ChatOpenAI 모델 초기화
llm = ChatOpenAI(
    model_name="gpt-4o-mini",
    temperature=0
    )

# 대화형 프롬프트 생성하기
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful chatbot"),
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)

# 대화 버퍼 메모리 생성하고, 메시지 반환 기능 활성화하기
memory = ConversationBufferMemory(return_messages=True, memory_key="chat_history")

# 요약 메모리로 교체할 경우
# memory = ConversationSummaryMemory(
#     llm=llm, return_messages=True, memory_key="chat_history"
# )

conversation_chain = MyConversationChain(llm, prompt, memory)
```

- 대화 시작

```
conversation_chain.invoke("안녕하세요? 만나서 반갑습니다. 제 이름은 Jay입니다.")
```

- 셀 출력 (1.0s)

```
'안녕하세요, Jay! 만나서 반갑습니다. 어떻게 도와드릴까요?'
```

```
conversation_chain.invoke("제 이름 기억해요?")
```

- 셀 출력 (1.0s)

```
'네, Jay님! 당신의 이름을 기억하고 있습니다. 어떻게 도와드릴까요?'
```

```
conversation_chain.invoke("앞으로는 영어로 말한 후 한국어로 번역해서 답변해주세요.")
```

- 셀 출력 (3.5s)

```
'Sure! I will respond in English first and then provide the translation in Korean. \n\n알겠습니다! 먼저 영어로 답변한 후
```

```
conversation_chain.invoke("제 이름을 멋있게 소개해주세요.")
```

- 셀 출력 (2.3s)

```
'Absolutely! Here’s a stylish introduction for you:\n\n"Meet Jay, a name that resonates with creativity and chari
```

```
conversation_chain.invoke("저에게 어울리는 영어이름을 3개 추천해주고, 그 이유도 말해주세요.")
```

- 셀 출력 (6.8s)

```
'Sure! Here are three English names that might suit you, along with the reasons for each:\n\n1. **Ethan**: This n
```

- memory 확인해보기

```
conversation_chain.memory.load_memory_variables({})["chat_history"]
```

- 셀 출력

```
[HumanMessage(content='안녕하세요? 만나서 반갑습니다. 제 이름은 Jay입니다.', additional_kwargs={}, response_metadata={}),
AIMessage(content='안녕하세요, Jay! 만나서 반갑습니다. 어떻게 도와드릴까요?', additional_kwargs={}, response_metadata={}),
HumanMessage(content='제 이름 기억해요?', additional_kwargs={}, response_metadata={}),
AIMessage(content='네, Jay님! 당신의 이름을 기억하고 있습니다. 어떻게 도와드릴까요?', additional_kwargs={}, response_metadata={
HumanMessage(content='앞으로는 영어로 말한 후 한국어로 번역해서 답변해주세요.', additional_kwargs={}, response_metadata={}),
AIMessage(content='Sure! I will respond in English first and then provide the translation in Korean. \n\n알겠습니다!
HumanMessage(content='제 이름을 멋있게 소개해주세요.', additional_kwargs={}, response_metadata={}),
AIMessage(content='Absolutely! Here’s a stylish introduction for you:\n\n"Meet Jay, a name that resonates with cr
HumanMessage(content='저에게 어울리는 영어이름을 3개 추천해주고, 그 이유도 말해주세요.', additional_kwargs={}, response_metadata
AIMessage(content='Sure! Here are three English names that might suit you, along with the reasons for each:\n\n1.
```

```
print(type(conversation_chain.memory.load_memory_variables({})))                    # <class 'dict'>
print(type(conversation_chain.memory.load_memory_variables({})["chat_history"]))    # <class 'list'>
```

```
# 보기 좋게 출력해보기

from langchain.schema import HumanMessage, AIMessage

for msg in conversation_chain.memory.load_memory_variables({})["chat_history"]:
    if isinstance(msg, HumanMessage):
        print("Human:", msg.content, "\n")
    elif isinstance(msg, AIMessage):
        print("AI:   ", msg.content, "\n")
```

- 셀 출력

Human: 안녕하세요? 만나서 반갑습니다. 제 이름은 Jay입니다.

AI: 안녕하세요, Jay! 만나서 반갑습니다. 어떻게 도와드릴까요?

Human: 제 이름 기억해요?

AI: 네, Jay님! 당신의 이름을 기억하고 있습니다. 어떻게 도와드릴까요?

Human: 앞으로는 영어로 말한 후 한국어로 번역해서 답변해주세요.

AI: Sure! I will respond in English first and then provide the translation in Korean.

알겠습니다! 먼저 영어로 답변한 후 한국어 번역을 제공하겠습니다.

Human: 제 이름을 멋있게 소개해주세요.

AI: Absolutely! Here's a stylish introduction for you:

"Meet Jay, a name that resonates with creativity and charisma. With a vibrant personality and a passion for explo

이제 한국어 번역입니다:

"제이를 소개합니다. 창의성과 매력이 어우러진 이름입니다. 활기찬 성격과 탐험에 대한 열정을 가진 제이는 가는 곳마다 오래도록 기억에 남는 인상을 남기는

Human: 저에게 어울리는 영어이름을 3개 추천해주고, 그 이유도 말해주세요.

AI: Sure! Here are three English names that might suit you, along with the reasons for each:

1. **Ethan**: This name means "strong" or "firm." It conveys a sense of reliability and determination, which migh

2. **Liam**: A name that means "strong-willed warrior." It has a modern and friendly vibe, suggesting someone who

3. **Noah**: This name means "rest" or "comfort." It has a gentle and calming quality, perfect for someone who br

이제 한국어 번역입니다:

물론입니다! 여기 당신에게 어울릴 수 있는 세 가지 영어 이름과 그 이유입니다:

1. **Ethan (이선)**: 이 이름은 "강한" 또는 "확고한"이라는 의미를 가지고 있습니다. 신뢰성과 결단력을 전달하며, 당신의 성격과 잘 어울릴 수 있습니

2. **Liam (리암)**: "강한 의지를 가진 전사"라는 의미를 가진 이름입니다. 현대적이고 친근한 느낌을 주며, 접근하기 쉽고 자신감 있는 사람을 암시합니

3. **Noah (노아)**: 이 이름은 "휴식" 또는 "편안함"이라는 의미를 가지고 있습니다. 부드럽고 차분한 특성을 지니고 있어, 주변 사람들에게 평화와 긍

LangGraph로 대화 흐름 관리 (최신 ver.)

- LangChain에서 deprecated(사용 중단 예정)
- langchain==1.0.0에서 **완전히 제거될 예정**
  - 이는 메모리 관리를 더 유연하고 효율적으로 처리하기 위한 변경
  - 대신 **trim_messages** 함수를 사용해 대화 히스토리를 토큰 제한에 맞게 잘라내는 방식으로 대체
  - invoke 대신 **LangGraph의 stream으로 대화 실행 = 히스토리 자동 유지**

왜 deprecated되었나?

- 기존 ConversationTokenBufferMemory는 대화의 최근 메시지를 토큰 제한에 맞게 유지했지만, **이제는 LangGraph나 LCEL (LangChain Expression Language)을 활용해 더 세밀한 제어가 가능**
- trim_messages를 사용하면 시스템 메시지를 유지하면서 최근 메시지만 토큰 제한 내로 유지

변화 내용

- `ConversationBufferMemory/ConversationSummaryMemory` 직접 꽂지 않음 → 대신 `BaseChatMessageHistory` 구현체
  - (예: `InMemoryChatMessageHistory`, `FileChatMessageHistory` 등)를 세션 단위로 생성 하여 **`RunnableWithMessageHistory`** 가 읽고/쓰기를 관리
  - **더이상 `memory.save_context` 수동 호출 X** → 히스토리는 **`RunnableWithMessageHistory`가 자동으로 업데이트함**
  - 세션 구분: **`config={"configurable":{"session_id":"..."}}` 로 전달 → 누락 시 에러, 오작동!!**
- `LLMChain/ConversationChain` 도 **폐기 예정 → `LCEL` 파이프라인 (`prompt | llm | parser`)** 을 **`RunnableWithMessageHistory`** 로 감싸서 사용
- 윈도우/토큰 제한 트리밍이 필요 하면 **`trim_messages` 유틸을 메시지 히스토리에 적용** 하거나 **`LangGraph` 요약/트리밍 전략 사용하기**

마이그레이션 관련 공식 가이드

- LangChain 메모리 마이그레이션 가이드
- 윈도우/토큰 메모리 대체-트리밍 전략과 요약 전략의 최신 권장 방식
- RunnableWithMessageHistory API 레퍼런스 - 입/출력 포맷, `session_id` 전달 규칙, 구성 필드 지정 등 상세한 내용 안내
- ConversationBufferMemory에서 RunnableWithMessageHistory/LangGraph로 이전하기

최신 방식으로 치환된 내용

- 변경_1
  - *삭제: `RunnablePassthrough.assign + RunnableLambda` 로 `memory.load_memory_variables` 를 주입하던 부분*
  - **변경: `MessagesPlaceholder("chat_history")` 만 남긴 뒤 `RunnableWithMessageHistory`가 자동 주입**
- 변경_2
  - *삭제: 커스텀 ConversationChain 클래스*
  - **변경: `LCEL 파이프라인(prompt | llm | parser)` 을 그대로 `RunnableWithMessageHistory` 로 감싸기**
- 변경_3
  - *삭제: 응답 후 수동 `memory.save_context` 호출*
  - **변경: `RunnableWithMessageHistory`가 처리**
- 변경_4: 윈도우 / 토큰 제한이 필요한 경우의 방법 변경
  - *변경 전: `ConversationBufferWindowMemory / ConversationTokenBufferMemory` 의 역할*
  - **변경:**
    - **`trim_messages` 유틸 등으로 대체 - 메시지 목록에 트리머를 적용하는 방법이 문서화되어 있음**
    - **`LangGrapph` 기반 요약 흐름 권장** - 대화가 길어져도 지연이 과도하지 않도록 점진 요약(요약 메시지를 유지)
- 변경_5: 파일 / 영구 저장이 필요할 때
  - *변경 전: `InMemoryChatMessageHistory`*
  - **변경: `FileChatMessageHistory` / `Redis`, `DB` 백엔드 구현체 권장**
    - `FileChatMessageHistory` → 세션별 `JSON` 파일에 기록 / 동일한 인터페이스로 교체 가능
    - 대규모 / 멀티-인스턴스 환경: `Redis`, `DB` 백엔드 구현체 사용 권장

```
# 최신 LangChain 메모리 방식: RunnableWithMessageHistory 사용 예시
"""
    - 기능: 세션 ID별로 대화 히스토리를 자동으로 로드/저장하여 이전 발화를 기억
    - 주의: 기존 ConversationBufferMemory 대신 BaseChatMessageHistory 계열을 사용
    - 요구 패키지: langchain, langchain-core, langchain-openai (최신 버전 권장)
"""

import os
from dotenv import load_dotenv
load_dotenv()                                   # True
```

```
# 1) 필수 임포트
from langchain_openai import ChatOpenAI                                          # 최신 네임스페이스 사용
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder  # LCEL 프롬프트
from langchain_core.output_parsers import StrOutputParser                    # 문자열 파서
from langchain_core.runnables import RunnableLambda, RunnablePassthrough    # LCEL 조합
from langchain_core.runnables.history import RunnableWithMessageHistory     # 최신 메모리 래퍼
from langchain_core.chat_history import InMemoryChatMessageHistory           # 기본 히스토리 구현


# 2) LLM 초기화
# .env 파일에서 환경변수 불러오기
load_dotenv()

api_key = os.getenv("OPENAI_API_KEY")

# OpenAI API 키 설정
openai.api_key = api_key

# LLM 초기화
llm = ChatOpenAI(
    temperature=0.7,                                          # 더 재미있는 예시를 위해 온도 높임
    openai_api_key=api_key,
    model="gpt-4o-mini"
)


# 3) 프롬프트 정의

# - chat_history 자리에 이전 대화가 자동 주입될 예정
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful chatbot."),               # 시스템 역할
        MessagesPlaceholder(variable_name="chat_history"),      # 메시지 히스토리 삽입 위치
        ("human", "{input}"),                                   # 사용자 입력
    ]
)


# 4) 기본 체인 구성 (프롬프트 → LLM → 문자열 파싱)

core_chain = prompt | llm | StrOutputParser()                   # LCEL 파이프라인


# 5) 세션별 메시지 히스토리 팩토리
# - session_id마다 별도 히스토리를 생성/관리
# - 운영환경에서는 FileChatMessageHistory/RedisChatMessageHistory 등으로 대체 가능

_session_store = {}                                                 # 간단한 예시용 (프로세스 메모리)
def get_session_history(session_id: str):
    # 한글 주석: 세션 ID로 히스토리를 조회/생성하여 반환
    if session_id not in _session_store:
        _session_store[session_id] = InMemoryChatMessageHistory()   # 기본 메모리형 히스토리
    return _session_store[session_id]                               # RunnableWithMessageHistory가 읽고/쓰기


# 6) RunnableWithMessageHistory로 래핑
# - input_messages_key: 입력 딕셔너리에서 사용자 메시지의 키 이름
# - history_messages_key: 프롬프트에서 MessagesPlaceholder로 받을 키 이름과 일치

with_history = RunnableWithMessageHistory(
    core_chain,
    get_session_history=get_session_history,            # 세션별 히스토리 팩토리
    input_messages_key="input",                         # 사용자 입력 키
    history_messages_key="chat_history",                # 프롬프트의 MessagesPlaceholder 키
)


# 7) 호출 예시
# - 각 호출 시 config.configurable.session_id를 반드시 전달
# - 동일한 session_id로 반복 호출하면 이전 대화를 기억

session_config = {"configurable": {"session_id": "user-123"}}                  # 세션 구분자
```

- 다른 예시로 대화 시도
    - **1995년에 태어났다** 정보를 히스토리에 저장
    - 두 번째 질문에서 AI가 2095년 미래 교통을 설명
    - 세 번째 질문에서 AI가 이전 연도 정보(1995) 를 기억해 **당신은 100살** 과 같은 답을 해야 히스토리가 제대로 작동한 것.

- 네 번째 질문으로 연속 맥락 유지까지 확인.

```
# ── 더 재미있는 예시 ──────────────────────────

session_config = {"configurable": {"session_id": "timetravel-001"}}

# 1) 사용자가 자신의 출생 연도를 알려 줌
ans1 = with_history.invoke(
    {"input": "안녕, 나는 1995년에 태어났어. 2095년의 세상은 어때?"},
    config=session_config
)
print("AI-1:", ans1)
```

- 셀 출력 (6.1s)

```
AI-1: 안녕하세요! 2095년의 세상은 여러 면에서 지금과 많이 다를 것이라고 예상됩니다. 기술이 엄청나게 발전해 있을 것이고, 인공지능과 로봇이 일상 생

1. **기술**: 가상현실(VR)과 증강현실(AR)이 더욱 발전하여, 사람들이 새로운 형태의 경험을 할 수 있을 것입니다. 인공지능은 개인 비서, 의료 진단, 

2. **환경**: 기후 변화에 대한 대응이 절실할 것으로 예상됩니다. 많은 나라가 지속 가능한 에너지와 교통수단을 채택하고 있을 것이고, 재생 가능 에너지원

3. **사회**: 인구 고령화 문제와 함께 다양한 사회적 변화가 있을 것입니다. 원격 근무가 더욱 보편화되고, 사람들의 삶의 방식이나 직업 형태도 많이 달라

4. **의료**: 의학 기술이 발전하여, 질병의 조기 진단과 맞춤형 치료가 가능해졌을 것입니다. 유전자 편집 기술이 더욱 발전하여, 유전적 질병을 예방할 수

물론, 이러한 예측은 현재의 기술 발전과 사회적 트렌드를 바탕으로 한 것이며, 실제로 어떤 모습일지는 여러 변수에 따라 달라질 수 있습니다. 당신의 생각은
```

```
# 2) AI가 미래 설명 → 사용자가 디테일 재확인
ans2 = with_history.invoke(
    {"input": "그럼 2095년에는 어떤 교통 수단이 대세야?"},
    config=session_config
)
print("AI-2:", ans2)
```

- 셀 출력 (6.1s)

```
AI-2: 2095년에는 교통 수단이 크게 발전하고 다양화될 것으로 예상됩니다. 몇 가지 주요 트렌드는 다음과 같을 것입니다:

1. **자율주행차**: 자율주행 기술이 보편화되어, 사람들은 직접 운전하는 대신 자율주행차에 탑승하여 목적지까지 이동할 수 있을 것입니다. 이는 교통사고를

2. **전기차와 수소차**: 화석 연료 기반의 차량은 거의 사라지고, 전기차와 수소차가 대세가 되어 있을 것입니다. 재생 가능 에너지를 활용한 충전 인프라가

3. **드론과 개인 항공기**: 드론을 이용한 물류 서비스가 발전하고, 개인용 드론이나 항공기가 도시 내 짧은 거리 이동을 위한 수단으로 등장할 가능성이 높

4. **고속 철도와 초고속 교통수단**: 초고속 열차(하이퍼루프와 같은 기술)가 대륙 간 이동을 혁신적으로 변화시킬 수 있으며, 지역 간 이동이 매우 빨라질

5. **대중교통의 혁신**: 대중교통 시스템이 스마트화되어, 실시간 데이터 분석을 통해 효율적인 운영이 이루어질 것입니다. 또한, 공유 교통 서비스가 더 널

6. **지속 가능한 교통수단**: 자전거 및 전동 킥보드와 같은 친환경 교통수단이 더욱 보편화되어, 도시 내 단거리 이동에 많이 사용될 것입니다.

이러한 변화들은 기술 발전, 환경 문제 해결 노력, 그리고 도시 설계와 관련된 사회적 요구에 따라 이루어질 것입니다. 당신은 어떤 교통 수단이 가장 기대되
```

```
# 3) 사용자가 '내가 몇 살이 되는지' 질문 → 이전 연도 정보 기억해야 답 가능
ans3 = with_history.invoke(
    {"input": "그 해에 나는 몇 살이지?"},
    config=session_config
)
print("AI-3:", ans3)
```

- 셀 출력 (1.1s)

```
AI-3: 2095년에는 당신이 100세가 됩니다. 1995년에 태어났으니까요! 축하할 만한 나이가 될 것 같습니다. 100세 생일을 멋지게 기념하는 방법을 생각하
```

```
# 4) 추가 기억 확인: 사용자가 시간 여행 일정 제안
ans4 = with_history.invoke(
    {"input": "2095년에 여행 가려면 지금부터 무엇을 준비하면 될까?"},
    config=session_config
)
print("AI-4:", ans4)
```

- 셀 출력 (6.4s)

  AI-4: 2095년에 여행을 가기 위해 지금부터 준비할 수 있는 몇 가지 사항이 있습니다. 미래의 여행 환경은 지금과 많이 다를 수 있으므로, 사전 준비가 중

  1. **언어 및 문화 학습**: 다양한 나라의 언어와 문화를 배우는 것은 여행을 더욱 풍부하게 만들어줍니다. 특히, 다문화 사회가 될 가능성이 높은 미래에서

  2. **기술 친화적 준비**: 미래의 여행에서는 기술이 큰 역할을 할 것입니다. 스마트폰, 웨어러블 기기, AR/VR 기기를 활용한 여행 경험을 위해 최신 기술

  3. **환경 친화적 여행 계획**: 지속 가능한 여행 방법을 고려하는 것이 중요해질 것입니다. 친환경적인 교통수단과 숙소를 선택하는 방법을 미리 알아보는 것

  4. **건강 관리 및 예방 접종**: 미래에는 새로운 질병이나 변종 바이러스가 등장할 수 있습니다. 그러므로 건강을 유지하고 필요한 예방 접종을 받는 것이

  5. **여행 보험**: 예상치 못한 상황에 대비해 여행 보험을 미리 준비하는 것이 좋습니다. 미래의 여행 환경에서는 안전이 더욱 중요할 수 있습니다.

  6. **여행 경로 및 숙소 연구**: 여행할 지역의 최신 정보를 미리 조사하고, 교통편과 숙소를 미리 예약해 두는 것이 좋습니다. 특히, 미래의 관광 명소나

  7. **재정 계획**: 여행 비용을 미리 계획하고 저축하는 것이 중요합니다. 미래의 물가 상승을 고려하여 예산을 세우는 것이 좋습니다.

  이러한 준비를 통해 2095년의 여행을 더욱 즐겁고 의미 있게 만드실 수 있을 것입니다. 어떤 지역이나 나라를 여행하고 싶은지 생각해 보셨나요?

- *next: SQLite 에 대화내용 저장*
