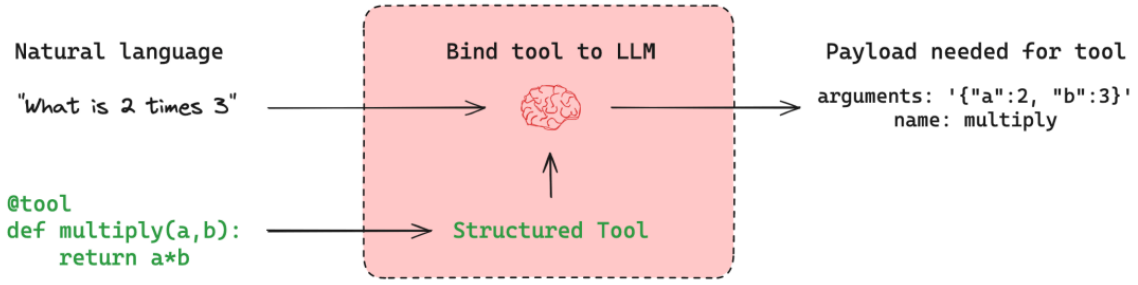
LangChain Tool Calling

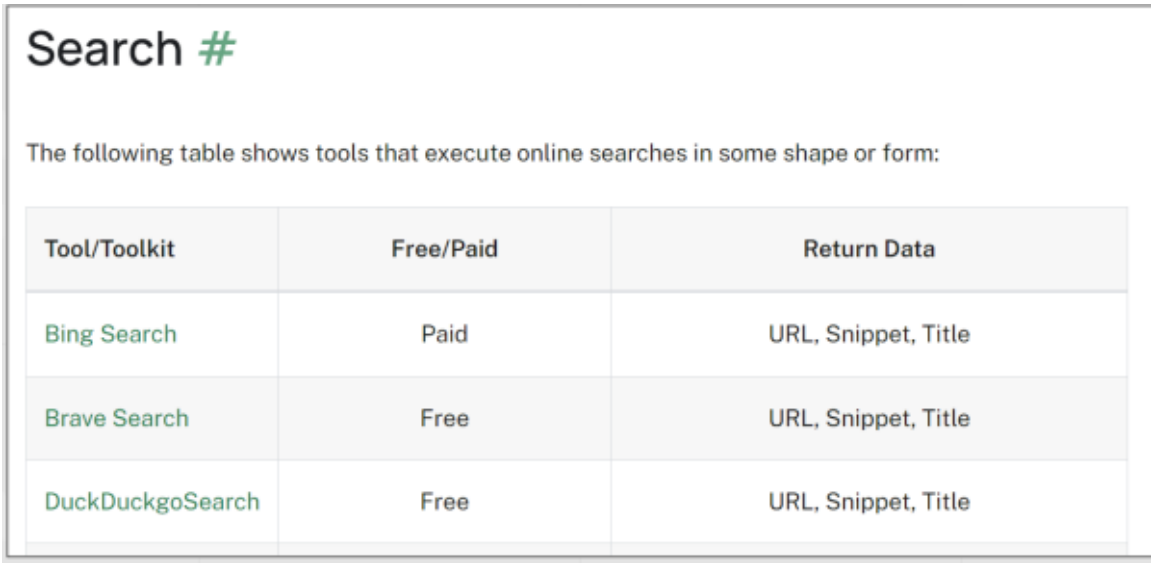
LangChain ToolCalling의 개요

- Toll Calling의 개념
  - LLM이 외부 기능이나 데이터에 접근할 수 있게 해주는 매커니즘
- 중요성
  - LLM의 한계 (최신 정보 부족, 특정 작업 수행 불가 등)를 극복하는 방법
- LLM과 외부 도구 연동의 필요성
  - 실시간 데이터 접근, 특수 기능 수행, 정확성 향상 등



LangChain에서 제공하는 내장 도구 (Tool)

- LangChain은 검색, 코드 인터프리터, 생산성 도구 등 다양한 Tool을 직접/제휴 형태로 제공
  - 검색 : DuckDuckGo, TavilySearch 등



Search

The following table shows tools that execute online searches in some shape or form:

Tool/Toolkit	Free/Paid	Return Data
Bing Search	Paid	URL, Snippet, Title
Brave Search	Free	URL, Snippet, Title
DuckDuckgoSearch	Free	URL, Snippet, Title

- 예시
  - Tavily 웹 검색 도구 사용
    - AI 기반 웹 검색 API를 제공하는 서비스
    - 인증키 : 환경 변수 TAVILY_API_KEY 설정

```
name:
tavily_search_results_json
--------------------------------------------------------------------
description:
A search engine optimized for comprehensive, accurate, and trusted
results. Useful for when you need to answer questions about current
events. Input should be a search query.
--------------------------------------------------------------------
args:
{'query': {'description': 'search query to look up', 'title': 'Query',
'type': 'string'}}
--------------------------------------------------------------------
```

  - Toll의 구성 요소
    - name : 도구의 이름
    - description : 도구가 수행하는 작업에 대한 설명
    - JSON schema : 도구의 입력을 정의하는 스키마
    - function : 실행할 함수 (선택적으로 비동기 함수도 가능)
- 도구 직접 실행

```
from langchain_community.tools import TavilySearchResult
s
```

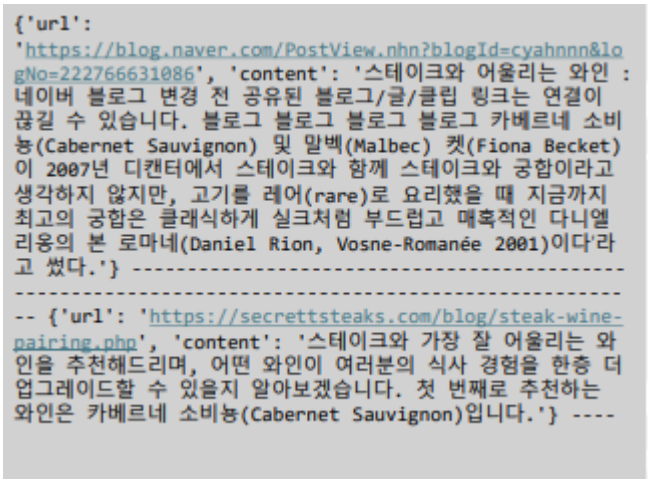
```
# 검색할 쿼리 설정 (query에 담긴 객체 그대로 웹 검색)
query = "스테이크와 어울리는 와인을 추천해주세요."

# Tavily 검색 도구 초기화 (최대 2개의 결과 반환)
web_search = TavilySearchResults(max_results = 2)

# 웹 검색 실행 (query 내용을 전달)
search_results = web_search.invoke(query)
```

- Tool Calling

```
from langchain_openai import ChatOpenAI

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model = 'gpt-4o-mini'

# 웹 검색 도구를 직접 LLM에 바인딩 가능
llm_with_tools = llm.bind_tools(tools=[web_search])

# 쿼리를 LLM에 전달해 결과 얻기 (LLM이 적절한 검색어를 생성)
ai_msg = llm_with.tools.invoke(query)
```

- Tool Calling 결과를 가지고 검색 작업 실행
  - args 스키마 사용

```
tool_call = ai_msg.tool_calls[0]
tool_output = web_search.invoke(tool_call["args"])
```



- tool_call 사용

```
tool_message = web_search.invoke(tool_call)
```

- Tool Message 정의

```
tool_message = ToolMessage(
    content = tool_output,
    tool_call_id = tool_call["id"],
    name = tool_call["name"]
)
```

ToolMessage(content=[{'url': 'https://secrettsteaks.com/blog/steak-wine-pairing.php', 'content': '스테이크와 가장 잘 어울리는 와인의 부드러운 맛은 스테이크의 풍미를 덮지 않고 자연스럽게 어우러져, 부담 없이 즐길 수 있는 조합을 만들어줍니다. 개인정보 처리 방침 개인정보 처리 방침 보기'}, {'url': 'https://blog.naver.com/PostView.nhn?blogId=cyahnnn&logNo=222766631086', 'content': '스테이크와 어울리는 와인 : 네이버 블로그 변경 전 공유된 블로그/글/클립 링크는 연결이 끊길 수 있습니다. 블로그 블로그 블로그 블로그 블로그 카베르네 소비뇽지금까지 최고의 궁합은 클래식하게 실크처럼 부드럽고 매혹적인 다니엘 리옹의 본 로마네(Daniel Rion, Vosne-Romanée 2001)이다'라고 썼다.'}], name='tavily_search_results_json', tool_call_id='call_51POU0gWyOJXqVR5PZsY6uPa')

ToolMessage를 LLM에 전달해 AI 답변 생성하기

- LLM 체인 정의

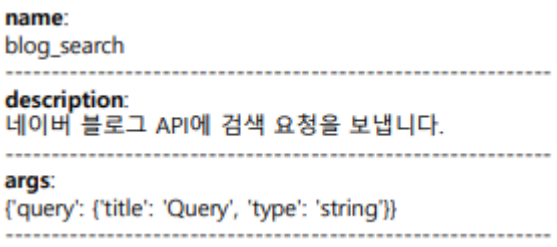
```
# 오늘 날짜 설정
today = datetime.today().strftime("%Y-%m-%d")

# 프롬프트 템플릿
prompt = ChatPromptTemplate([
    ("system", f"You are a helpful AI assistant"),
    ("system", f"Today's date is {today}.")<
    ("placeholder", "{messages}"),
])

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model = "gpt-4o-mini")

# LLM에 도구를 바인딩
llm_with_tools = llm.bind_tools(tools = [web_search_too
l])

# LLM 체인 생성
llm_chain = prompt | llm_with_tools
```

- LLM 체인에 ToolMessage 전달

```
@chain
def web_search_chain(user_input: str, config: RunnableCo
nfig):
    input_ = {"user_input": user_input}
    ai_msg = llm_chain.invoke(input_, config = config)
    tool_msgs = web_search_tool.batch)
        ai_msg.tool_calls, config = config)
    return llm_chain.invoke(
        {**input_, "messages": [ai_msg, *tool_msgs]},
        config = config
        )
# 체인 실행
web_search_chain.invoke("오늘 모엣샹동의 가격은 얼마인가요?")
```

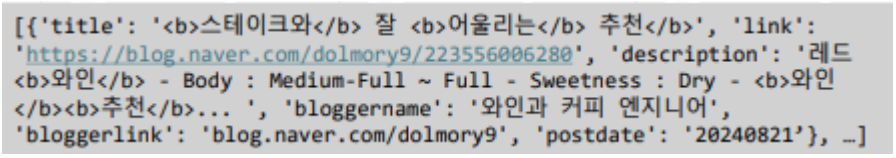
사용자 정의 도구 (Tool)

- LangChain은 사용자가 직접 도구를 정의해 사용하는 방법을 제공
- 가장 대표적인 방법 : @tool 데코레이터 사용
  - 함수를 LangChain 도구로 변환하는 방법
  - 도구 함수 작성 가이드라인
    - 명확한 입출력 정의, 단일 책임 원칙 준수
  - 도구 설명(Description) 작성
    - LLM이 도구의 기능을 정확히 이해하고 사용하도록 작성




```
from langchain_core.tools import tool

@tool
```

```
def blog_search(query: str) -> List[Dict]:
    # tool의 description이 됨
    """네이버 블로그 API에 검색 요청을 보냅니다."""
    url = "https://openapi.naver.com/v1/search/blog.jso
n"
    headers = {
        "X-Naver-Client-Id": NAVER_CLIENT_ID,
        "X-Naver-Client-Secret": NAVER_CLIENT_SECET
    }
    # display = 검색 결과 갯수
    params = {"query": query, "display": 10, "start": 1}
    response = requests.get(url, headers = headers,
        params = params)
    if response.status_code == 200:
        return response.json()['items']
    else:
        return []
# query = args 부분이 됨
query = "스테이크와 어울리는 와인을 추천해주세요."
search_results = blog_search.invoke(query)
```



Runnable 객체를 도구(Tool) 변환

- 문자열이나 dict 입력을 받는 Runnable을 도구로 변환
- as_tool 사용

```
from langchain_community.document_loaders import WikipediaL
oader
from langchain_core.documents import Document
from langchain_core.runnables import RunnableLambda
from pydantic import BaseModel, Field
```

```
from typing import List

# WikipediaLoader를 사용하여 위키피디아 문서를 검색하는 함수
def search_wiki(input_data: dict) -> List[Document]:
    """Search Wikipedia documents based on user input (quer
y) and return k documents"""
    query = input_data["query"]
    k = input_data.get("k", 2)
    wiki_loader = WikipediaLoader(query=query, load_max_doc
s=k, lang="ko")
    wiki_docs = wiki_loader.load()
    return wiki_docs

# 도구 호출에 사용할 입력 스키마 정의
class WikiSearchSchema(BaseModel):
    """Input schema for Wikipedia search."""
    query: str = Field(..., description="The query to searc
h for in Wikipedia")
    k: int = Field(2, description="The number of documents
to return (default is 2)")

# RunnableLambda 함수를 사용하여 위키피디아 문서 로더를 Runnable
로 변환
runnable = RunnableLambda(search_wiki)
wiki_search = runnable.as_tool(
    name="wiki_search",
    description=dedent("""
        Use this tool when you need to search for informati
on on Wikipedia.
        It searches for Wikipedia articles related to the u
ser's query and returns
        a specified number of documents. This tool is usefu
l when general knowledge
        or background information is required.
    """),
    args_schema=WikiSearchSchema
)
```

- k값 : 몇 개의 값을 가져올 것인지
- wikipedia는 default language가 english이기 때문에 lang이라는 매개 변수로 설정해줘야
- description, name 등을 사용해 세부적인 설정을 해줄 수 있음
- 입력 스키마를 상세히 설정해야 도구의 성능을 높일 수 있음

```
# 위키 검색 실행
query = "파스타의 유래"
wiki_results = wiki_search.invoke({"query":query})

# 검색 결과 출력
for result in wiki_results:
    print(result)
    print("-" * 100)
```

- 바인딩해서 2개의 도구를 사용
- 사용자 쿼리에 따라 도구 호출이 필요하면 호출

```
# LLM에 도구를 바인딩 (2개의 도구 바인딩)
llm_with_tools = llm.bind_tools(tools=[search_web, wiki_sea
rch])

# 도구 호출이 필요한 LLM 호출을 수행
query = "서울 강남의 유명한 파스타 맛집은 어디인가요? 그리고 파스타
의 유래를 알려주세요. "
ai_msg = llm_with_tools.invoke(query)

# LLM의 전체 출력 결과 출력
pprint(ai_msg)
print("-" * 100)

# 메시지 content 속성 (텍스트 출력)
pprint(ai_msg.content)
print("-" * 100)

# LLM이 호출한 도구 정보 출력
```

```
pprint(ai_msg.tool_calls)
print("-" * 100)
```

LCEL 체인

- 위키피디아 문서를 검색하고 내용 요약하는 체인
- 그대로 출력하는게 아닌 요약해서 출력하도록

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda
from langchain_community.document_loaders import WikipediaL
oader

# WikipediaLoader를 사용하여 위키피디아 문서를 검색하고 텍스트로
반환하는 함수
def wiki_search_and_summarize(input_data: dict):
    wiki_loader = WikipediaLoader(query=input_data["quer
y"], load_max_docs=2, lang="ko")
    wiki_docs = wiki_loader.load()

    formatted_docs =[
        f'<Document source="{doc.metadata["source"]}"/>\n{d
oc.page_content}\n</Document>'
        for doc in wiki_docs
        ]

    return formatted_docs

# 요약 프롬프트 템플릿
summary_prompt = ChatPromptTemplate.from_template(
    "Summarize the following text in a concise manner:\n\n
{context}\n\nSummary:"
)

# LLM 및 요약 체인 설정
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
summary_chain = (
```

```
    {"context": RunnableLambda(wiki_search_and_summarize)}
    | summary_prompt | llm | StrOutputParser()
)

# 요약 테스트
summarized_text = summary_chain.invoke({"query":"파스타의 유
래"})
pprint(summarized_text)
```

- LLM 모델에게 요약 역할을 부여함
- List 축약 문법을 사용해 HTML 태그 형태로 formatting
- summary 프롬프트를 생성 → LLM에 연결 → LLM 결과를 파싱해 출력

```
# 도구 호출에 사용할 입력 스키마 정의
class WikiSummarySchema(BaseModel):
    """Input schema for Wikipedia search."""
    query: str = Field(..., description="The query to searc
h for in Wikipedia")

# as_tool 메소드를 사용하여 도구 객체로 변환
wiki_summary = summary_chain.as_tool(
    name="wiki_summary",
    description=dedent("""
        Use this tool when you need to search for informati
on on Wikipedia.
        It searches for Wikipedia articles related to the u
ser's query and returns
        a summarized text. This tool is useful when general
knowledge
        or background information is required.
    """),
    args_schema=WikiSummarySchema
)

# 도구 속성
print("자료형: ")
print(type(wiki_summary))
```

```
print("-"*100)

print("name: ")
print(wiki_summary.name)
print("-"*100)

print("description: ")
pprint(wiki_summary.description)
print("-"*100)

print("schema: ")
pprint(wiki_summary.args_schema.schema())
print("-"*100)
```

- 입력 스키마를 정의하고, 도구를 생성

```
# LLM에 도구를 바인딩
llm_with_tools = llm.bind_tools(tools=[search_web, wiki_sum
mary])

# 도구 호출이 필요한 LLM 호출을 수행
query = "서울 강남의 유명한 파스타 맛집은 어디인가요? 그리고 파스타
의 유래를 알려주세요. "
ai_msg = llm_with_tools.invoke(query)

# LLM의 전체 출력 결과 출력
pprint(ai_msg)
print("-" * 100)

# 메시지 content 속성 (텍스트 출력)
pprint(ai_msg.content)
print("-" * 100)

# LLM이 호출한 도구 정보 출력
pprint(ai_msg.tool_calls)
print("-" * 100)
```

- 웹 검색 도구와 wiki 요약 도구를 바인딩해서 호출

- LLM 도구 자체가 웹 검색 호출 여부를 판단하고 실행

```
from datetime import datetime
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableConfig, chain

# 오늘 날짜 설정
today = datetime.today().strftime("%Y-%m-%d")

# 프롬프트 템플릿
prompt = ChatPromptTemplate([
    ("system", f"You are a helpful AI assistant. Today's da
te is {today}."),
    ("human", "{user_input}"),
    ("placeholder", "{messages}"),
])

# LLM에 도구를 바인딩
llm_with_tools = llm.bind_tools(tools=[wiki_summary])

# LLM 체인 생성
llm_chain = prompt | llm_with_tools

# 도구 실행 체인 정의
@chain
def wiki_summary_chain(user_input: str, config: RunnableCon
fig):
    input_ = {"user_input": user_input}
    ai_msg = llm_chain.invoke(input_, config=config)
    print("ai_msg: \n", ai_msg)
    print("-"*100)
    tool_msgs = wiki_summary.batch(ai_msg.tool_calls, confi
g=config)
    print("tool_msgs: \n", tool_msgs)
    print("-"*100)
    return llm_chain.invoke({**input_, "messages": [ai_msg,
*tool_msgs]}, config=config)
```

```
# 체인 실행
response = wiki_summary_chain.invoke("파스타의 유래에 대해서 알
려주세요.")

# 응답 출력
pprint(response.content)
```

벡터 저장소 검색기

- @tool decorator 사용
- text loader를 사용해 문서를 읽음

```
from langchain.document_loaders import TextLoader

# 메뉴판 텍스트 데이터를 로드
loader = TextLoader("./data/restaurant_menu.txt", encoding
="utf-8")
documents = loader.load()

print(len(documents))
```

- 정규 표현식을 정의해 모델이 이해할 수 있는 형태로 만들어줌

```
from langchain_core.documents import Document

# 문서 분할 (Chunking)
def split_menu_items(document):
    """
    메뉴 항목을 분리하는 함수
    """
    # 정규표현식 정의
    pattern = r'(\d+\.\s.*?)(?=\n\n\d+\.|$)'
    menu_items = re.findall(pattern, document.page_content,
re.DOTALL)

    # 각 메뉴 항목을 Document 객체로 변환
    menu_documents = []
```

```
    for i, item in enumerate(menu_items, 1):
        # 메뉴 이름 추출
        menu_name = item.split('\n')[0].split('.', 1)[1].st
rip()

        # 새로운 Document 객체 생성
        menu_doc = Document(
            page_content=item.strip(),
            metadata={
                "source": document.metadata['source'],
                "menu_number": i,
                "menu_name": menu_name
            }
        )
        menu_documents.append(menu_doc)

    return menu_documents


# 메뉴 항목 분리 실행
menu_documents = []
for doc in documents:
    menu_documents += split_menu_items(doc)

# 결과 출력
print(f"총 {len(menu_documents)}개의 메뉴 항목이 처리되었습니
다.")
for doc in menu_documents[:2]:
    print(f"\n메뉴 번호: {doc.metadata['menu_number']}")
    print(f"메뉴 이름: {doc.metadata['menu_name']}")
    print(f"내용:\n{doc.page_content[:100]}...")
```

- 10개의 chunk를 verctorstore에 chromaDB를 사용해 저장
- Ollama Embedding model 사용

```
from langchain_core.documents import Document

# 문서 분할 (Chunking)
```

```
def split_menu_items(document):
    """
    메뉴 항목을 분리하는 함수
    """
    # 정규표현식 정의
    pattern = r'(\d+\.\s.*?)(?=\n\n\d+\.|$)'
    menu_items = re.findall(pattern, document.page_content,
re.DOTALL)

    # 각 메뉴 항목을 Document 객체로 변환
    menu_documents = []
    for i, item in enumerate(menu_items, 1):
        # 메뉴 이름 추출
        menu_name = item.split('\n')[0].split('.', 1)[1].st
rip()

        # 새로운 Document 객체 생성
        menu_doc = Document(
            page_content=item.strip(),
            metadata={
                "source": document.metadata['source'],
                "menu_number": i,
                "menu_name": menu_name
            }
        )
        menu_documents.append(menu_doc)

    return menu_documents


# 메뉴 항목 분리 실행
menu_documents = []
for doc in documents:
    menu_documents += split_menu_items(doc)

# 결과 출력
print(f"총 {len(menu_documents)}개의 메뉴 항목이 처리되었습니
다.")
```

```
for doc in menu_documents[:2]:
    print(f"\n메뉴 번호: {doc.metadata['menu_number']}")
    print(f"메뉴 이름: {doc.metadata['menu_name']}")
    print(f"내용:\n{doc.page_content[:100]}...")
```

- Verctor Store 사전 준비

```
# Chroma Vectorstore를 사용하기 위한 준비
from langchain_chroma import Chroma
from langchain_ollama  import OllamaEmbeddings

embeddings_model = OllamaEmbeddings(model="bge-m3")

# Chroma 인덱스 생성
menu_db = Chroma.from_documents(
    documents=menu_documents,
    embedding=embeddings_model,
    collection_name="restaurant_menu",
    persist_directory="./chroma_db",
)

# Retriever 생성
menu_retriever = menu_db.as_retriever(
    search_kwargs={'k': 2},
)

# 쿼리 테스트
query = "시그니처 스테이크의 가격과 특징은 무엇인가요?"
docs = menu_retriever.invoke(query)
print(f"검색 결과: {len(docs)}개")

for doc in docs:
    print(f"메뉴 번호: {doc.metadata['menu_number']}")
    print(f"메뉴 이름: {doc.metadata['menu_name']}")
    print()
```

- Wine 메뉴 처리

```
# 와인 메뉴 텍스트 데이터를 로드
loader = TextLoader("./data/restaurant_wine.txt", encoding
="utf-8")
documents = loader.load()

# 메뉴 항목 분리 실행
menu_documents = []
for doc in documents:
    menu_documents += split_menu_items(doc)

# 결과 출력
print(f"총 {len(menu_documents)}개의 메뉴 항목이 처리되었습니
다.")
for doc in menu_documents[:2]:
    print(f"\n메뉴 번호: {doc.metadata['menu_number']}")
    print(f"메뉴 이름: {doc.metadata['menu_name']}")
    print(f"내용:\n{doc.page_content[:100]}...")


# Chroma 인덱스 생성
wine_db = Chroma.from_documents(
    documents=menu_documents,
    embedding=embeddings_model,
    collection_name="restaurant_wine",
    persist_directory="./chroma_db",
)

wine_retriever = wine_db.as_retriever(
    search_kwargs={'k': 2},
)

query = "스테이크와 어울리는 와인을 추천해주세요."
docs = wine_retriever.invoke(query)
print(f"검색 결과: {len(docs)}개")

for doc in docs:
    print(f"메뉴 번호: {doc.metadata['menu_number']}")
```

```
    print(f"메뉴 이름: {doc.metadata['menu_name']}")
    print()
```

- 도구 (Tool) 정의

```
# 벡터 저장소 로드 (Chroma 도구 사용, 동일 embedding 모델 사용)
menu_db = Chroma(
    embedding_function=embeddings_model,
    collection_name="restaurant_menu",
    persist_directory="./chroma_db",
)

# Similarity Search를 사용해 유사한 결과를 가져옴
@tool
def search_menu(query: str) -> List[Document]:
        # 역할 정의
    """
    Securely retrieve and access authorized restaurant menu
information from the encrypted database.
    Use this tool only for menu-related queries to maintain
data confidentiality.
    """
    docs = menu_db.similarity_search(query, k=2)
    if len(docs) > 0:
        return docs

    return [Document(page_content="관련 메뉴 정보를 찾을 수 없
습니다.")]

# 도구 속성
print("자료형: ")
print(type(search_menu))
print("-"*100)

print("name: ")
print(search_menu.name)
print("-"*100)
```

```
print("description: ")
pprint(search_menu.description)
print("-"*100)

print("schema: ")
pprint(search_menu.args_schema.schema())
print("-"*100)
```

```
# LLM에 도구를 바인딩 (2개의 도구 바인딩)
llm_with_tools = llm.bind_tools(tools=[search_menu, search_
wine])

# 도구 호출이 필요한 LLM 호출을 수행
query = "시그니처 스테이크의 가격과 특징은 무엇인가요? 그리고 스테이
크와 어울리는 와인 추천도 해주세요."
ai_msg = llm_with_tools.invoke(query)

# LLM의 전체 출력 결과 출력
pprint(ai_msg)
print("-" * 100)

# 메시지 content 속성 (텍스트 출력)
pprint(ai_msg.content)
print("-" * 100)

# LLM이 호출한 도구 정보 출력
pprint(ai_msg.tool_calls)
print("-" * 100)
```

- 여러 개의 도구(Tool) 호출하기
  - 웹 검색 | wikipedia 검색 요약 | 와인 메뉴 검색 | 레스토랑 음식 메뉴 검색

```
tools = [search_web, wiki_summary, search_wine, search_men
u]
for tool in tools:
    print(tool.name)
```

```
from datetime import datetime
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableConfig, chain

# 오늘 날짜 설정
today = datetime.today().strftime("%Y-%m-%d")

# 프롬프트 템플릿
prompt = ChatPromptTemplate([
    ("system", f"You are a helpful AI assistant. Today's da
te is {today}."),
    ("human", "{user_input}"),
    ("placeholder", "{messages}"),
])

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model="gpt-4o-mini")

# 4개의 검색 도구를 LLM에 바인딩
llm_with_tools = llm.bind_tools(tools=tools)

# LLM 체인 생성
llm_chain = prompt | llm_with_tools

# 도구 실행 체인 정의
@chain
def restaurant_menu_chain(user_input: str, config: Runnable
Config):
    input_ = {"user_input": user_input}
    ai_msg = llm_chain.invoke(input_, config=config)

    tool_msgs = []
    for tool_call in ai_msg.tool_calls:
        print(f"{tool_call['name']}: \n{tool_call}")
        print("-"*100)

        if tool_call["name"] == "search_web":
            tool_message = search_web.invoke(tool_call, con
```

```
fig=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "wiki_summary":
            tool_message = wiki_summary.invoke(tool_call, c
onfig=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "search_wine":
            tool_message = search_wine.invoke(tool_call, co
nfig=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "search_menu":
            tool_message = search_menu.invoke(tool_call, co
nfig=config)
            tool_msgs.append(tool_message)

    print("tool_msgs: \n", tool_msgs)
    print("-"*100)
    return llm_chain.invoke({**input_, "messages": [ai_msg,
*tool_msgs]}, config=config)

# 체인 실행
response = restaurant_menu_chain.invoke("시그니처 스테이크의
가격과 특징은 무엇인가요? 그리고 스테이크와 어울리는 와인 추천도 해주
세요.")

# 응답 출력
print(response.content)
```

```
# 체인 실행
response = restaurant_menu_chain.invoke("파스타 메뉴가 있나요?
이 음식의 역사 또는 유래를 알려주세요.")

# 응답 출력
print(response.content)
```

Few-Shot 프롬프팅을 활용한 ToolCalling 성능 개선

- Few-Shot 포롬프팅
  - 모델에게 몇 가지 예시를 제공해 원하는 출력 형식이나 작업 수행 방식을 보여주는 기법
  - 모델에게 도구를 어떻게 사용해야 하는지 예시를 통해 보여주는 기법으로 사용
- Tool Calling 적용 과정
  1. 예시 생성
  2. 프롬프트 템플릿 생성
  3. 프롬프트 생성 및 모델 호출
  4. 응답 파싱

Few-Shot 프롬프팅

- 각 도구의 용도를 구분해 few-shot 예제로 제시
- Few-Shot 도구 호출

```
from langchain_core.messages import AIMessage, HumanMessag
e, ToolMessage
from langchain_core.prompts import ChatPromptTemplate

examples = [
    HumanMessage("트러플 리조또의 가격과 특징, 그리고 어울리는 와
인에 대해 알려주세요.", name="example_user"),
    AIMessage("메뉴 정보를 검색하고, 위키피디아에서 추가 정보를 찾
은 후, 어울리는 와인을 검색해보겠습니다.", name="example_assistan
t"),
    AIMessage("", name="example_assistant", tool_calls=[{"n
ame": "search_menu", "args": {"query": "트러플 리조또"}, "i
d": "1"}]),
    ToolMessage("트러플 리조또: 가격 ₩28,000, 이탈리아 카나롤리
쌀 사용, 블랙 트러플 향과 파르메산 치즈를 듬뿍 넣어 조리", tool_cal
l_id="1"),
    AIMessage("트러플 리조또의 가격은 ₩28,000이며, 이탈리아 카나
롤리 쌀을 사용하고 블랙 트러플 향과 파르메산 치즈를 듬뿍 넣어 조리합
```

```
니다. 이제 추가 정보를 위키피디아에서 찾아보겠습니다.", name="example_assistant"),
    AIMessage("", name="example_assistant", tool_calls=[{"name": "wiki_summary", "args": {"query": "트러플 리조또", "k": 1}, "id": "2"}]),
    ToolMessage("트러플 리조또는 이탈리아 요리의 대표적인 리조또 요리 중 하나로, 고급 식재료인 트러플을 사용하여 만든 크리미한 쌀 요리입니다. 주로 아르보리오나 카나롤리 등의 쌀을 사용하며, 트러플 오일이나 생 트러플을 넣어 조리합니다. 리조또 특유의 크리미한 질감과 트러플의 강렬하고 독특한 향이 조화를 이루는 것이 특징입니다.", tool_call_id="2"),
    AIMessage("트러플 리조또의 특징에 대해 알아보았습니다. 이제 어울리는 와인을 검색해보겠습니다.", name="example_assistant"),
    AIMessage("", name="example_assistant", tool_calls=[{"name": "search_wine", "args": {"query": "트러플 리조또에 어울리는 와인"}, "id": "3"}]),
    ToolMessage("트러플 리조또와 잘 어울리는 와인으로는 주로 중간 바디의 화이트 와인이 추천됩니다. 1. 샤르도네: 버터와 오크향이 트러플의 풍미를 보완합니다. 2. 피노 그리지오: 산뜻한 산미가 리조또의 크리미함과 균형을 이룹니다. 3. 베르나차: 이탈리아 토스카나 지방의 화이트 와인으로, 미네랄리티가 트러플과 잘 어울립니다.", tool_call_id="3"),
    AIMessage("트러플 리조또(₩28,000)는 이탈리아의 대표적인 리조또 요리 중 하나로, 이탈리아 카나롤리 쌀을 사용하고 블랙 트러플 향과 파르메산 치즈를 듬뿍 넣어 조리합니다. 주요 특징으로는 크리미한 질감과 트러플의 강렬하고 독특한 향이 조화를 이루는 점입니다. 고급 식재료인 트러플을 사용해 풍부한 맛과 향을 내며, 주로 아르보리오나 카나롤리 등의 쌀을 사용합니다. 트러플 리조또와 잘 어울리는 와인으로는 중간 바디의 화이트 와인이 추천됩니다. 특히 버터와 오크향이 트러플의 풍미를 보완하는 샤르도네, 산뜻한 산미로 리조또의 크리미함과 균형을 이루는 피노 그리지오, 그리고 미네랄리티가 트러플과 잘 어울리는 이탈리아 토스카나 지방의 베르나차 등이 좋은 선택이 될 수 있습니다.", name="example_assistant"),
]

system = """You are an AI assistant providing restaurant menu information and general food-related knowledge.
For information about the restaurant's menu, use the search
```

```
_menu tool.
For other general information, use the wiki_summary tool.
For wine recommendations or pairing information, use the se
arch_wine tool.
If additional web searches are needed or for the most up-to
-date information, use the search_web tool.
"""

few_shot_prompt = ChatPromptTemplate.from_messages([
    ("system", system),
    *examples,
    ("human", "{query}"),
])

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model="gpt-4o-mini")

# 검색 도구를 직접 LLM에 바인딩 가능
llm_with_tools = llm.bind_tools(tools=tools)

# Few-shot 프롬프트를 사용한 체인 구성
fewshot_search_chain = few_shot_prompt | llm_with_tools

# 체인 실행
query = "스테이크 메뉴가 있나요? 스테이크와 어울리는 와인을 추천해주
세요."
response = fewshot_search_chain.invoke(query)

# 결과 출력
for tool_call in response.tool_calls:
    print(tool_call)
```

```
# 체인 실행
query = "파스타의 유래에 대해서 알고 있나요? 서울 강남의 파스타 맛
집을 추천해주세요."
response = fewshot_search_chain.invoke(query)

# 결과 출력
```

```
for tool_call in response.tool_calls:
    print(tool_call)
```

답변 생성 체인

```
from datetime import datetime
from langchain_core.messages import AIMessage, HumanMessag
e, ToolMessage
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableConfig, chain
from langchain_openai import ChatOpenAI

# 오늘 날짜 설정
today = datetime.today().strftime("%Y-%m-%d")

# 프롬프트 템플릿
system = """You are an AI assistant providing restaurant me
nu information and general food-related knowledge.
For information about the restaurant's menu, use the search
_menu tool.
For other general information, use the wiki_summary tool.
For wine recommendations or pairing information, use the se
arch_wine tool.
If additional web searches are needed or for the most up-to
-date information, use the search_web tool.
"""

few_shot_prompt = ChatPromptTemplate.from_messages([
    ("system", system + f"Today's date is {today}."),
    *examples,
    ("human", "{user_input}"),
    ("placeholder", "{messages}"),
])

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model="gpt-4o-mini")
```

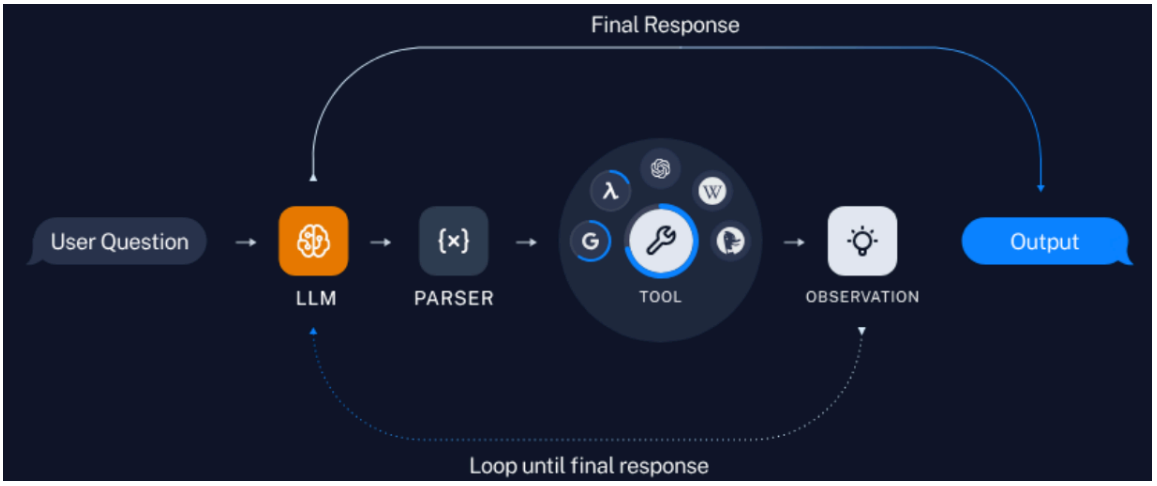
```
# 검색 도구를 직접 LLM에 바인딩 가능
llm_with_tools = llm.bind_tools(tools=tools)

# Few-shot 프롬프트를 사용한 체인 구성
fewshot_search_chain = few_shot_prompt | llm_with_tools

# 도구 실행 체인 정의
@chain
def restaurant_menu_chain(user_input: str, config: Runnable
Config):
    input_ = {"user_input": user_input}
    ai_msg = llm_chain.invoke(input_, config=config)

    tool_msgs = []
    for tool_call in ai_msg.tool_calls:
        print(f"{tool_call['name']}: \n{tool_call}")
        print("-"*100)

        if tool_call["name"] == "search_web":
            tool_message = search_web.invoke(tool_call, con
fig=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "wiki_summary":
            tool_message = wiki_summary.invoke(tool_call, c
onfig=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "search_wine":
            tool_message = search_wine.invoke(tool_call, co
nfig=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "search_menu":
            tool_message = search_menu.invoke(tool_call, co
nfig=config)
            tool_msgs.append(tool_message)
```

```
    print("tool_msgs: \n", tool_msgs)
    print("-"*100)
    return fewshot_search_chain.invoke({**input_, "message
s": [ai_msg, *tool_msgs]}, config=config)


# 체인 실행
query = "스테이크 메뉴가 있나요? 스테이크와 어울리는 와인을 추천해주
세요."
response = restaurant_menu_chain.invoke(query)

# 응답 출력
pprint(response.content)
```

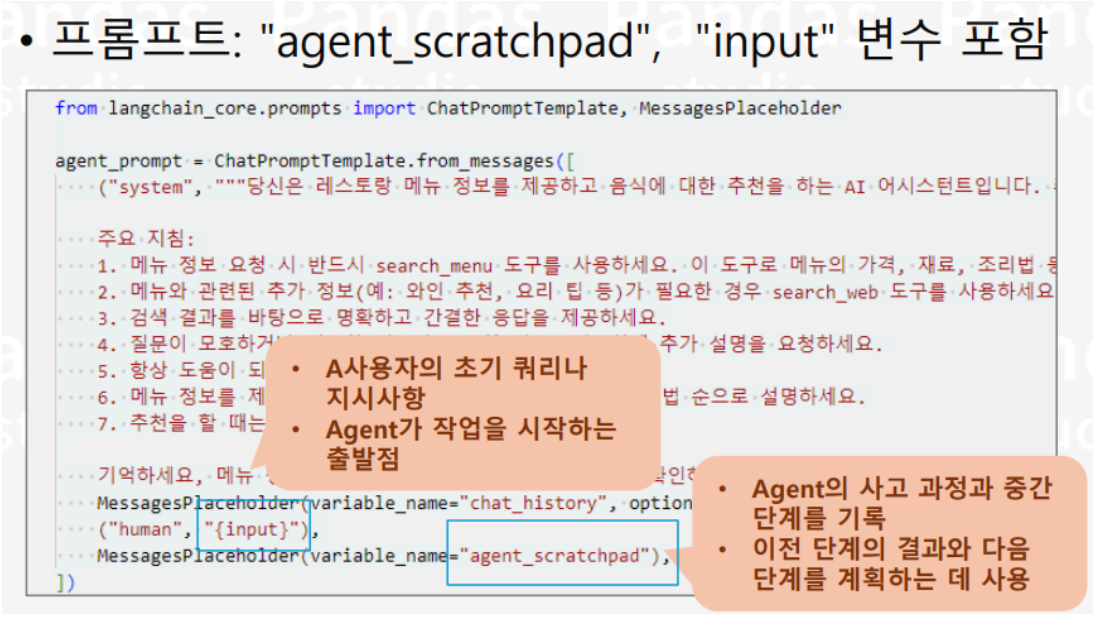
```
# 체인 실행
query = "파스타의 유래에 대해서 알고 있나요? 서울 강남의 파스타 맛
집을 추천해주세요."
response = restaurant_menu_chain.invoke(query)

# 응답 출력
pprint(response.content)
```

LangChain Agent 개요

- LLM을 추론 엔진으로 사용해 어떤 행동을 할지, 그 행동의 입력은 무엇일지 결정하는 시스템
- 언어 모델이 단순히 텍스트를 출력하는 것을 넘어 실제 행동을 취하게 함
- 행동의 결과를 다시 Agent에 피드백하여 추가 행동할지 또는 작업을 완료할 지를 결정



- langchain 라이브러리에서 제공하는 2개의 방법
  - create_tool_calling_agent



  - create_tool_calling_agent 함수 사용

```
from langchain.agents import create_tool_calling_agen
t

tools = [web_search_tool, blog_search, search_menu]
agent = create_tool_calling_agent(llm, tools, agent_p
rompt)
```

```
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent = agent, tools =
tools, verbose=True)

query = "시그니처 스테이크의 가격과 특징은 무엇인가요? 그리고
스테이크와 어울리는
와인 추천도 해주세요."

agent_response = agent_executor.invoke({"input": quer
y})
```

```
from langchain_core.prompts import ChatPromptTemplate, Mess
agesPlaceholder

agent_prompt = ChatPromptTemplate.from_messages([
    ("system", dedent("""
        You are an AI assistant providing restaurant menu i
nformation and general food-related knowledge.
        Your main goal is to provide accurate information a
nd effective recommendations to users.

        Key guidelines:
        1. For restaurant menu information, use the search_
menu tool. This tool provides details on menu items, includ
ing prices, ingredients, and cooking methods.
        2. For general food information, history, and cultu
ral background, utilize the wiki_summary tool.
        3. For wine recommendations or food and wine pairin
g information, use the search_wine tool.
        4. If additional web searches are needed or for the
most up-to-date information, use the search_web tool.
        5. Provide clear and concise responses based on the
search results.
        6. If a question is ambiguous or lacks necessary in
formation, politely ask for clarification.
        7. Always maintain a helpful and professional tone.
```

```
        8. When providing menu information, describe in the
order of price, main ingredients, and distinctive cooking m
ethods.
        9. When making recommendations, briefly explain the
reasons.
        10. Maintain a conversational, chatbot-like style i
n your final responses. Be friendly, engaging, and natural
in your communication.


        Remember, understand the purpose of each tool accur
ately and use them in appropriate situations.
        Combine the tools to provide the most comprehensive
and accurate answers to user queries.
        Always strive to provide the most current and accur
ate information.
        """)),
    MessagesPlaceholder(variable_name="chat_history", optio
nal=True),
    ("human", "{input}"),
    MessagesPlaceholder(variable_name="agent_scratchpad"),
])
```

```
# Tool calling Agent 생성
from langchain.agents import AgentExecutor, create_tool_cal
ling_agent

tools = [search_web, wiki_summary, search_wine, search_men
u]
agent = create_tool_calling_agent(llm, tools, agent_prompt)

# AgentExecutor 생성
agent_executor = AgentExecutor(agent=agent, tools=tools, ve
rbose=True)
```

```
# AgentExecutor 실행


```

```
query = "시그니처 스테이크의 가격과 특징은 무엇인가요? 그리고 스테이
크와 어울리는 와인 추천도 해주세요."
agent_response = agent_executor.invoke({"input": query})
```

Gradio

- 머신러닝이나 인공지능 애플리케이션을 빠르게 웹으로 구현할 수 있음
- chat_history를 통해 최근 메시지 기반 별도 응답 가능
-

```
import gradio as gr
from typing import List, Tuple

def answer_invoke(message: str, history: List[Tuple[str, st
r]]) -> str:
    try:
        # 채팅 기록을 AI에게 전달할 수 있는 형식으로 변환
        chat_history = []
        for human, ai in history:
            chat_history.append(HumanMessage(content=huma
n))
            chat_history.append(AIMessage(content=ai))

        # agent_executor를 사용하여 응답 생성
        response = agent_executor.invoke({
            "input": message,
            "chat_history": chat_history[-2:]    # 최근 2개
의 메시지 기록만을 활용
        })

        # agent_executor의 응답에서 최종 답변 추출
        return response['output']
    except Exception as e:
        # 오류 발생 시 사용자에게 알리고 로그 기록
        print(f"Error occurred: {str(e)}")
        return "죄송합니다. 응답을 생성하는 동안 오류가 발생했습니
```

```
다. 다시 시도해 주세요."

# 예제 질문 정의
example_questions = [
    "시그니처 스테이크의 가격과 특징을 알려주세요.",
    "트러플 리조또와 잘 어울리는 와인을 추천해주세요.",
    "해산물 파스타의 주요 재료는 무엇인가요? 서울 강남 지역에 레스토
랑을 추천해주세요.",
    "채식주의자를 위한 메뉴 옵션이 있나요?"
]

# Gradio 인터페이스 생성
demo = gr.ChatInterface(
    fn=answer_invoke,
    title="레스토랑 메뉴 AI 어시스턴트",
    description="메뉴 정보, 추천, 음식 관련 질문에 답변해 드립니
다.",
    examples=example_questions,
    theme=gr.themes.Soft()
)

# 데모 실행
demo.launch()
```

```
# 데모 종료
demo.close()
```