

# AI Agent 과정

Agent 기초

## 목차 Agent 기초

---

1. AI Agent 개요
2. Tool의 개념과 설계
3. 다양한 문서 처리 Tool 구현

## 2. Tool의 개념과 설계



- Agent

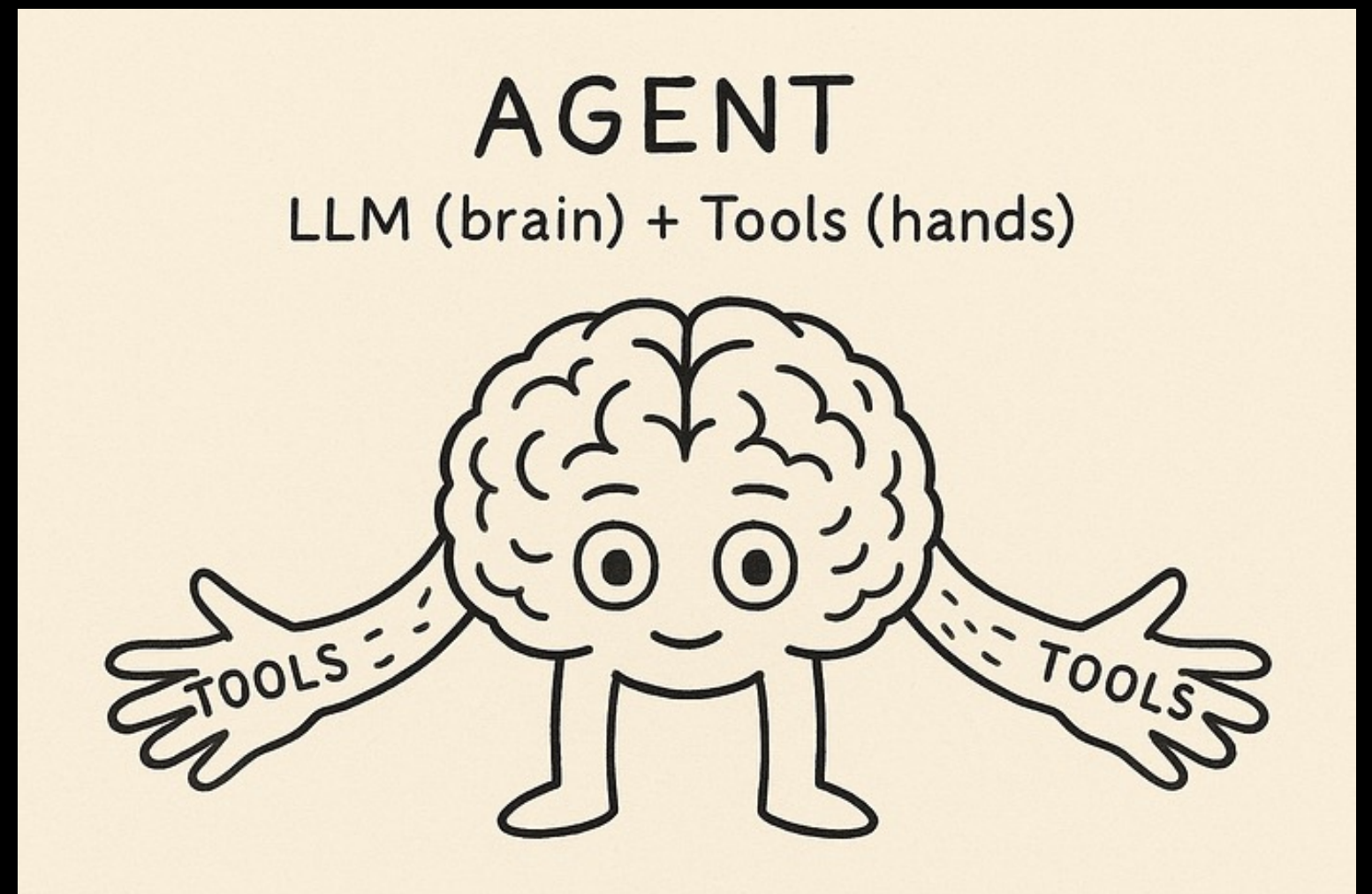
- 주변 환경을 인식하고, 정보를 처리하고, 결정을 내리고, 목표를 달성하기 위해 **행동**하는 소프트웨어 기반 시스템

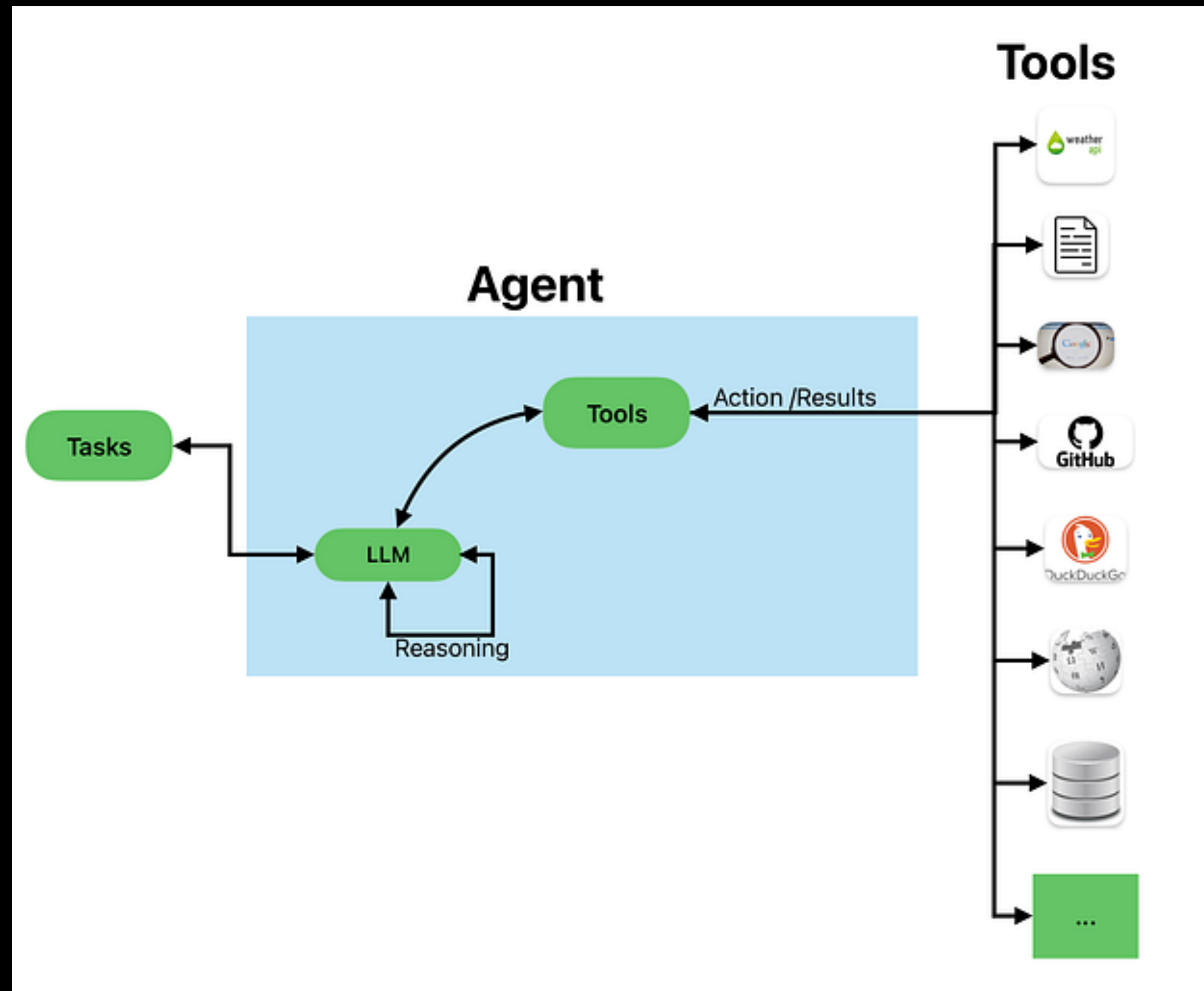
- 행동 (Actions)

- 도구(Tools)를 통해 이루어짐

- 도구 (Tools)

- LLM에게 제공하는 명확한 목적을 수행하는 함수
- Agent가 외부 세계와 상호작용하기 위한 인터페이스







## Tool 사용 중요성

- 기능 확장
  - LLM이 자체적으로 가지고 있지 않은 정보나 기능에 접근할 수 있도록 함
- 실시간 정보 접근
  - 웹 검색 도구를 통해 최신 정보를 얻을 수 있어, 신뢰도 높은 답변을 얻을 수 있음
- 정확성 향상
  - 계산기 도구나 데이터베이스 쿼리 도구를 통해 정확한 수치나 데이터를 얻을 수 있음
- 다양한 작업 수행
  - 코드 실행, 파일 조작, API 호출 등 다양한 작업을 수행



## Tool 사용 과정

- 도구 정의
  - 사용할 도구들을 정의하고 설정
- 프롬프트 설계
  - 모델이 도구를 적절히 사용할 수 있도록 프롬프트 설계
- 모델 추론
  - 언어 모델이 주어진 태스크를 분석하고 필요한 도구 결정
- 도구 호출
  - 모델이 선택한 도구를 호출하고 결과를 받음
- 결과 통합
  - 도구 호출의 결과를 모델의 응답에 통합





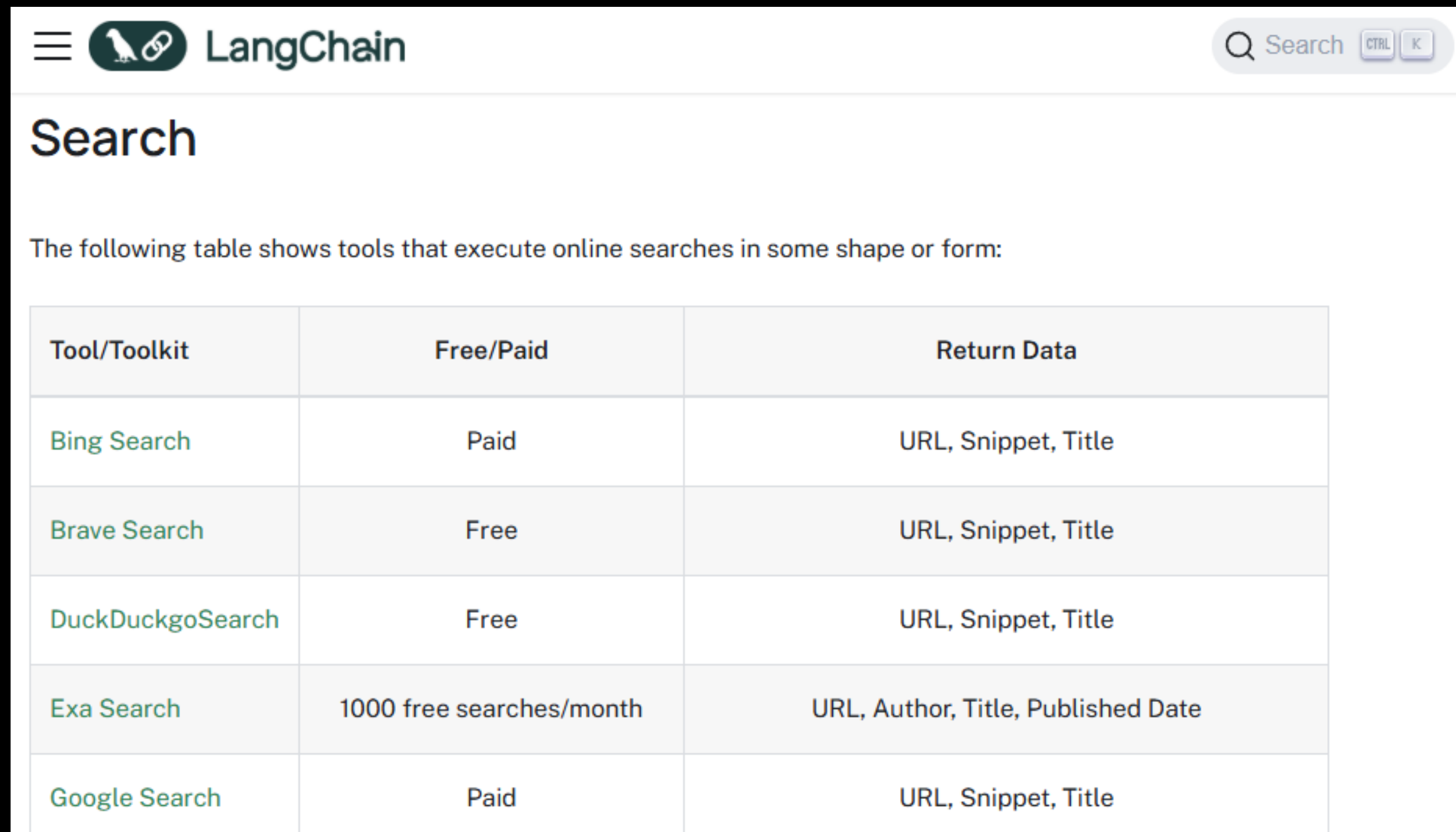
## Tool 필수 요소

- 이름 (str)
  - 도구의 고유한 식별자로 각 도구가 구별되고 쉽게 참조될 수 있도록 함
- Description (str)
  - 도구가 수행하는 작업에 대한 간략한 설명으로 도구 기능과 사용 방법을 이해하는 데 중요한 맥락 제공
- Args\_schema (pydantic.BaseModel)
  - Tool이 받는 입력값의 구조와 데이터 타입을 정의
- Return\_direct (boolean)
  - True로 설정하면 Agent는 도구를 호출한 후 프로세스를 중지하고 결과를 사용자에게 직접 반환



# Built-in Tools

- LangChain 같은 플랫폼에서 기본적으로 제공하는 사전에 정의된 도구(tool)와 툴킷(toolkit)
- Tool은 단일 도구, Toolkit은 여러 도구를 묶어서 하나의 도구로 사용하는 것



The screenshot shows the LangChain website's search results page. The header includes the LangChain logo and a search bar. The main heading is "Search". Below it, a text block states: "The following table shows tools that execute online searches in some shape or form:". A table follows, listing various search tools with their pricing and return data.

Tool/Toolkit	Free/Paid	Return Data
Bing Search	Paid	URL, Snippet, Title
Brave Search	Free	URL, Snippet, Title
DuckDuckgoSearch	Free	URL, Snippet, Title
Exa Search	1000 free searches/month	URL, Author, Title, Published Date
Google Search	Paid	URL, Snippet, Title



## Third-party Tool

- 외부 서비스 업체가 만든 API나 기능을 Agent에 연결해서 쓰는 도구(tool)
- LangChain에서 기본 제공하는 건 아니지만 외부 SDK나 API로 쉽게 연결 가능
- 별도의 API Key 또는 인증 설정 필요
- 서비스 제공 범위나 요금 정책 확인 필요



# LangChain Tool 사용

- 사용 예시 : Wikipedia integration

```
!pip install -qU langchain-community wikipedia
```

```
from langchain_community.tools import WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper

api_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max=100)
tool = WikipediaQueryRun(api_wrapper=api_wrapper)

print(tool.invoke({"query": "langchain"}))
```

Page: LangChain

Summary: LangChain is a framework designed to simplify the creation of applications



# LangChain Tool 사용

- 도구의 기본 속성 확인

```
print(f"Name: {tool.name}")  
print(f"Description: {tool.description}")  
print(f"args schema: {tool.args}")  
print(f"returns directly?: {tool.return_direct}")
```

```
Name: wikipedia  
Description: A wrapper around Wikipedia. Useful for when you need to answer general questions about people, place  
args schema: {'query': {'description': 'query to look up on wikipedia', 'title': 'Query', 'type': 'string'}}  
returns directly?: False
```





# LangChain Tool 사용

- 사용자 도구 정의 (Customizing Default Tools)
  - 내장된 이름, 설명, 인수의 JSON 스키마를 수정할 수 있음
  - 인수의 JSON 스키마를 정의할 때 입력 값들이 함수와 동일하게 유지되어야 함
  - 따라서 입력 매개변수 자체는 변경하면 안 되지만, 각 입력에 대한 커스텀 설명은 쉽게 정의할 수 있음



# LangChain Tool 사용

- 사용자 도구 정의 (Customizing Default Tools)
  - 1단계 : 커스텀 입력 스키마 정의

```
from pydantic import BaseModel, Field

class WikiInputs(BaseModel):
    """Inputs to the wikipedia tool."""
    query: str = Field(
        description="query to look up in Wikipedia, should be 3 or less words"
    )
```



# LangChain Tool 사용

- 사용자 도구 정의 (Customizing Default Tools)
- 2단계 : 커스터마이징된 도구 생성

```
tool = WikipediaQueryRun(  
    name="wiki-tool",                # 도구 이름 변경  
    description="look up things in wikipedia", # 설명 변경  
    args_schema=WikiInputs,          # 커스텀 스키마 적용  
    api_wrapper=api_wrapper,  
    return_direct=True,              # 결과를 직접 반환  
)
```



# LangChain Toolkits 사용

- Toolkits

- 특정 작업을 위해 함께 사용되도록 설계된 도구들의 컬렉션
- 일반적으로 `get_tools()` 메서드로 도구 목록을 검색
- Toolkits을 사용하면 관련된 여러 도구를 한 번에 로드하고 사용할 수 있어 특정 도메인 작업을 효율적으로 수행할 수 있음

```
# 툴킷 초기화
toolkit = ExampleToolkit(...)

# 도구 리스트 가져오기
tools = toolkit.get_tools()
```





# LangChain Toolkits 사용

- Toolkits 사용의 장점
  - 관련 도구들의 그룹화
    - 특정 도메인(예: Gmail, GitHub, SQL 등)에 필요한 도구들이 하나의 패키지로 제공
  - 편리한 초기화
    - 개별 도구들을 하나씩 설정할 필요 없이 Toolkit 하나로 필요한 도구를 한 번에 사용 가능
  - 일관된 인터페이스
    - 모든 Toolkit이 동일한 `get_tools()` 패턴을 따름



# LangChain Toolkits 사용

- GmailToolkit 사용 예시

```
# Gmail 툴킷 사용 예시
from langchain_community.agent_toolkits import GmailToolkit

# 툴킷 초기화 (인증 정보 포함)
gmail_toolkit = GmailToolkit()

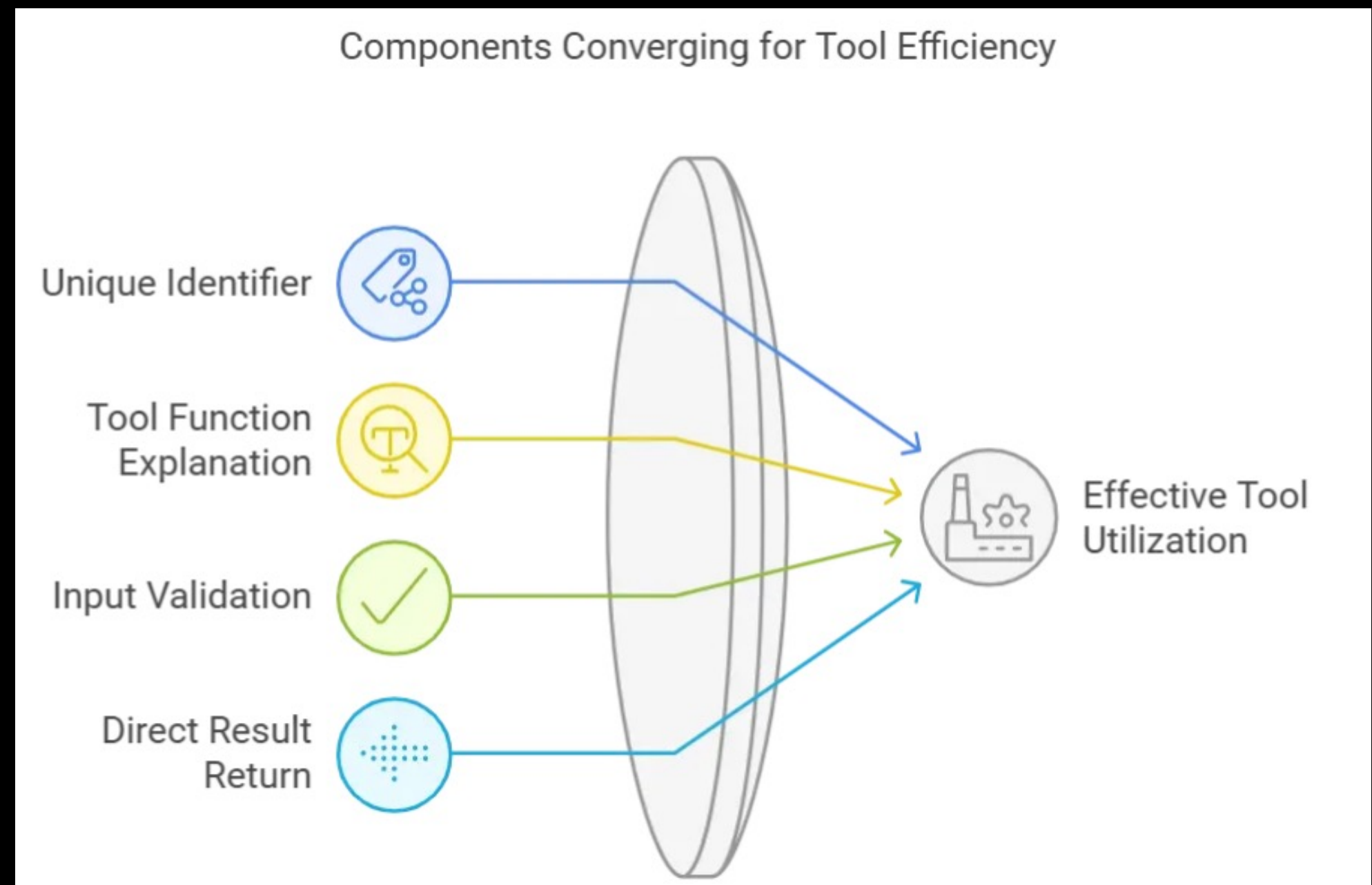
# 사용 가능한 Gmail 관련 도구들 가져오기
gmail_tools = gmail_toolkit.get_tools()

# 도구들 확인
for tool in gmail_tools:
    print(f"Tool: {tool.name} - {tool.description}")
```



# Custom Tool

- LangChain에서 제공하는 built-in 도구 외에 사용자가 직접 도구를 정의하여 사용할 수 있음
- 내부 시스템에 연동이 가능
- 보안/권한 제어를 직접 설정 가능





# Custom Tool 제작 방법

- Custom Tool 제작을 위한 4가지 방법
  1. @tool 데코레이터 사용
  2. StructuredTool Class 사용
  3. BaseTool Class 사용
  4. LangChain Runnable 사용





# @tool 데코레이터

## • 장점

- LangChain 도구를 정의하는 가장 간단하고 직관적인 방법
- 함수를 래핑하여 도구를 빠르게 생성 가능
- 독스트링(docstring)을 활용해 설명(Description) 작성이 용이함

## • 단점

- 복잡한 도구 제작에는 한계가 있음
- 데코레이터가 함수를 LangChain과 강하게 결합시켜, 독립적인 테스트나 재사용성 면에서 유연성이 떨어짐

```
from langchain.tools import tool
import requests

@tool
def zipcode_info(zipcode: str) -> str:
    """Fetches information related to a zipcode"""

    url = f"https://api.zippopotam.us/us/{zipcode}"

    payload = {}
    headers = {}

    response = requests.request("GET", url, headers=headers, data=payload)
    return f"the information about {zipcode} is : {response.text}"
```



# StructuredTool Class

- 장점

- 구조화된 입력 매개변수가 필요하거나 유효성 검사가 필요한 도구에 적합
- 서브클래싱 없이 StructuredTool.from\_function 메서드로 도구 정의 가능
- 복잡한 도구의 관리와 확장이 용이함
- Pydantic 스키마를 통한 향상된 유효성 검사 지원

- 단점

- BaseTool 대비 제한된 커스터마이징 (완전한 제어 기능 부족)
- 간단한 도구에는 과도하게 복잡할 수 있음

```
from langchain.tools import StructuredTool
from pydantic import BaseModel
import requests

class ZipcodeInput ( BaseModel ):
    zipcode: str

def zipcode_info ( zipcode: str ) -> str :

    """우편번호와 관련된 정보를 가져옵니다"""
    url = f"https://api.zippopotam.us/us/ {zipcode} "

    payload = {}
    headers = {}

    response = requests.request( "GET" , url, headers=headers, data=payload)
    return f" {zipcode} 에 대한 정보 는 : {response.text} 입니다 "
```



# StructuredTool Class

- Pydantic
  - 타입 힌트를 사용해 데이터 검증과 파싱을 수행하는 라이브러리
  - 사전에 입력 매개변수의 타입을 지정
  - 입력값이 올바른지 자동으로 체크
  - 잘못된 입력이 들어왔을 때 자동으로 에러 발생

```
class ZipcodeInput ( BaseModel ):
    zipcode: str
```



# BaseTool Class

- 장점

- 가장 유연하고 커스터마이징이 가능한 옵션, 도구 기능에 대한 완전한 제어권 제공
- 비동기 기능, 상태 관리, 외부 API 통합 등 고급 기능을 모두 지원
- 내부 상태를 유지하고 관리 가능

- 단점

- LangChain 내부 구조에 대한 깊은 이해와 많은 코드 작성 필요
- 높은 학습 곡선으로 전문 지식을 갖춘 사용자에게 적합

```
from langchain.tools import BaseTool
from typing import Type
from pydantic import BaseModel
import requests

class ZipcodeInput(BaseModel):
    zipcode: str

class CustomZipcodeTool(BaseTool):
    name: str = "zip_info_tool"
    description: str = "Fetches information related to a zipcode"
    args_schema: Type[BaseModel] = ZipcodeInput

    def _run(self, zipcode: str):
        url = f"https://api.zippopotam.us/us/{zipcode}"
        payload = {}
        headers = {}
        response = requests.request("GET", url, headers=headers, data=payload)
        return f"The information about {zipcode} is: {response.text}"

    async def _arun(self, zipcode: str):
        url = f"https://api.zippopotam.us/us/{zipcode}"
        payload = {}
        headers = {}
        response = requests.request("GET", url, headers=headers, data=payload)
        return f"(async) The information about {zipcode} is: {response.text}"
```





# Runnable Interface

- 장점

- 여러 컴포넌트 체이닝하고 통합하는 데 이상적
- LangChain 체이닝 방식을 그대로 활용하여 도구 생성 가능
- 동기 및 비동기 워크플로우 모두 지원

- 단점

- 간단한 도구 제작 시 @tool 이나 StructuredTool 보다 복잡함
- LangChain 체이닝 방식에 대한 이해 필요

```
from langchain_core.language_models import GenericFakeChatModel
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages(
    [("human", "Hello. Please respond in the style of {answer_style}.")]
)

# Placeholder LLM
llm = GenericFakeChatModel(messages=iter(["hello matey"]))

chain = prompt | llm | StrOutputParser()

as_tool = chain.as_tool(
    name="Style responder", description="Responds in a specific style based on"
)
```



# Description

- Agent가 언제 어떤 Tool을 사용할지 결정하는 데 사용됨
- 잘 작성된 Description은 Agent 성능 향상을 크게 향상

## Wikipedia 툴 예시:

```
class WikipediaQueryRun(BaseTool):  
    name: str = "wikipedia"  
    description: str = (  
        "A wrapper around Wikipedia. "  
        "Useful for when you need to answer general questions about "  
        "people, places, companies, facts, historical events, or other subjects. "  
        "Input should be a search query."  
    )
```

## Golden Query 툴 예시:

```
class GoldenQueryRun(BaseTool):  
    name: str = "golden_query"  
    description: str = (  
        "A wrapper around Golden Query API."  
        " Useful for getting entities that match"  
        " a natural language query from Golden's Knowledge Base."  
        "\nExample queries:"  
        "\n- companies in nanotech"  
        "\n- list of cloud providers starting in 2019"  
        "\nInput should be the natural language query."  
        "\nOutput is a paginated list of results or an error object"  
        " in JSON format."  
    )
```



# Description

- Description 주요 구성 요소
  - 도구 기능 설명
    - 해당 Tool이 어떤 기능을 수행하는 지 요약
    - 예 : Searches web or images via Google Search API
  - 사용 용도 설명
    - Tool이 유용한 상황이나 목적 설명
    - 예 : Useful for finding recent news or online resources
  - 입력 요구사항
    - Tool 사용 시 필수적으로 제공해야 하는 입력값과 그 형식
    - 예 : Input should be a search query



# Description

- Description 주요 구성 요소
  - 출력/반환값 설명
    - Tool 실행 후 반환되는 결과 형식 및 내용 설명
    - 예 : Returns JSON with metadata and result items (title, snippet, link)
  - 제한사항
    - Tool이 처리할 수 없는 범위나 상황 명시
    - 예 : Works only within set search scope and quota limits
  - 데이터 소스
    - Tool이 참조하거나 호출하는 외부 데이터나 API
    - 예 : Uses Google Custom Search JSON API





# Description

- Description 주요 구성 요소
  - 에러 처리 방법
    - Tool 동작 중 오류가 발생했을 때의 처리 규칙
    - 예 : Returns error message for auth failure or quota exceeded
  - 샘플 질의
    - Tool이 처리 가능한 입력 예시 제공
    - 예 : "query": "latest AI news"



## Description 작성 Tip

- 명시적으로 작성
  - 무엇을 하는지, 언제 사용해야 하는지, 언제 사용하지 말아야 하는지 명확히 작성
- 사용하지 말아야 하는 경우도 포함
  - 'when not to use it'을 추가하면 툴 남용을 방지할 수 있음
- temperature=0 설정
  - 랜덤성을 줄여 모델이 설명을 엄격하게 따르도록 함



## Description 작성 Tip

- Tool 우선순위 지정
  - 유사한 tool이 여러 개 있을 때, 특정 상황에서 하나를 우선 사용하도록 지정할 수 있음
  - 설명에 다음과 같은 문구 포함
    - Use this tool more than regular search if you are asked factual information about a person, place, or thing
    - Use this tool more than the Wikipedia tool if you are asked about current events, recent information, or news



## Description 작성 Tip

- 음악 관련 Query에 Music Search Tool을 Search Tool 보다 우선 사용하도록 명시한 예시

```
tools = [
    Tool(
        name="Search",
        func=search.run,
        description="Ask the targeted prompts to get answers about recent affairs",
    ),
    Tool(
        name="Music Search",
        func=lambda x: "Mariah Carey's song called All I Want For Christmas",
        description="helpful in searching music and should be used more than the other
tools for queries related to Music, like 'tell me the most viewed song in 2022' or
'about the singer of a song'",
    ),
]

agent.run("what is the most famous song of Christmas")
# 결과: All I Want For Christmas Is You
```