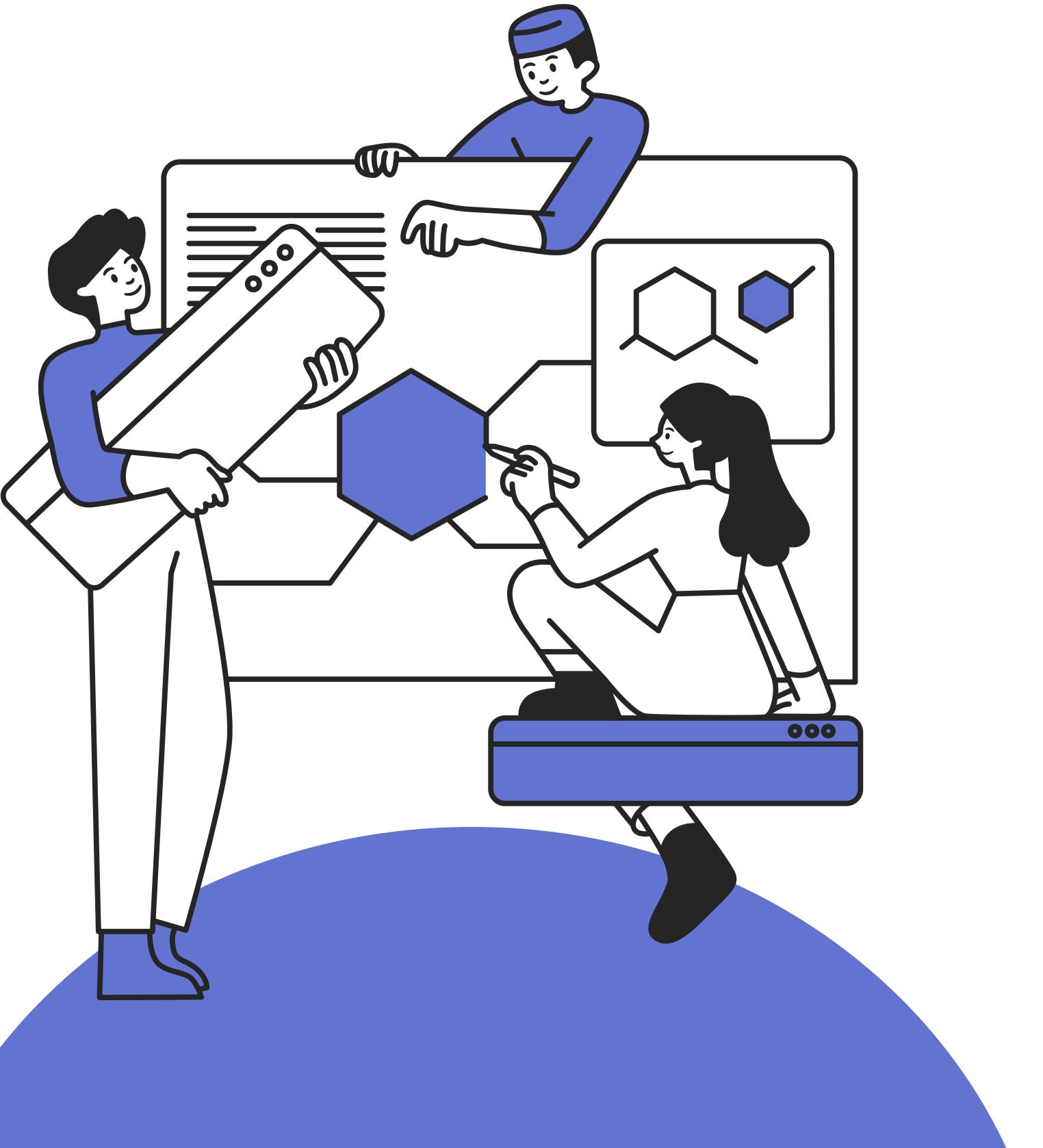
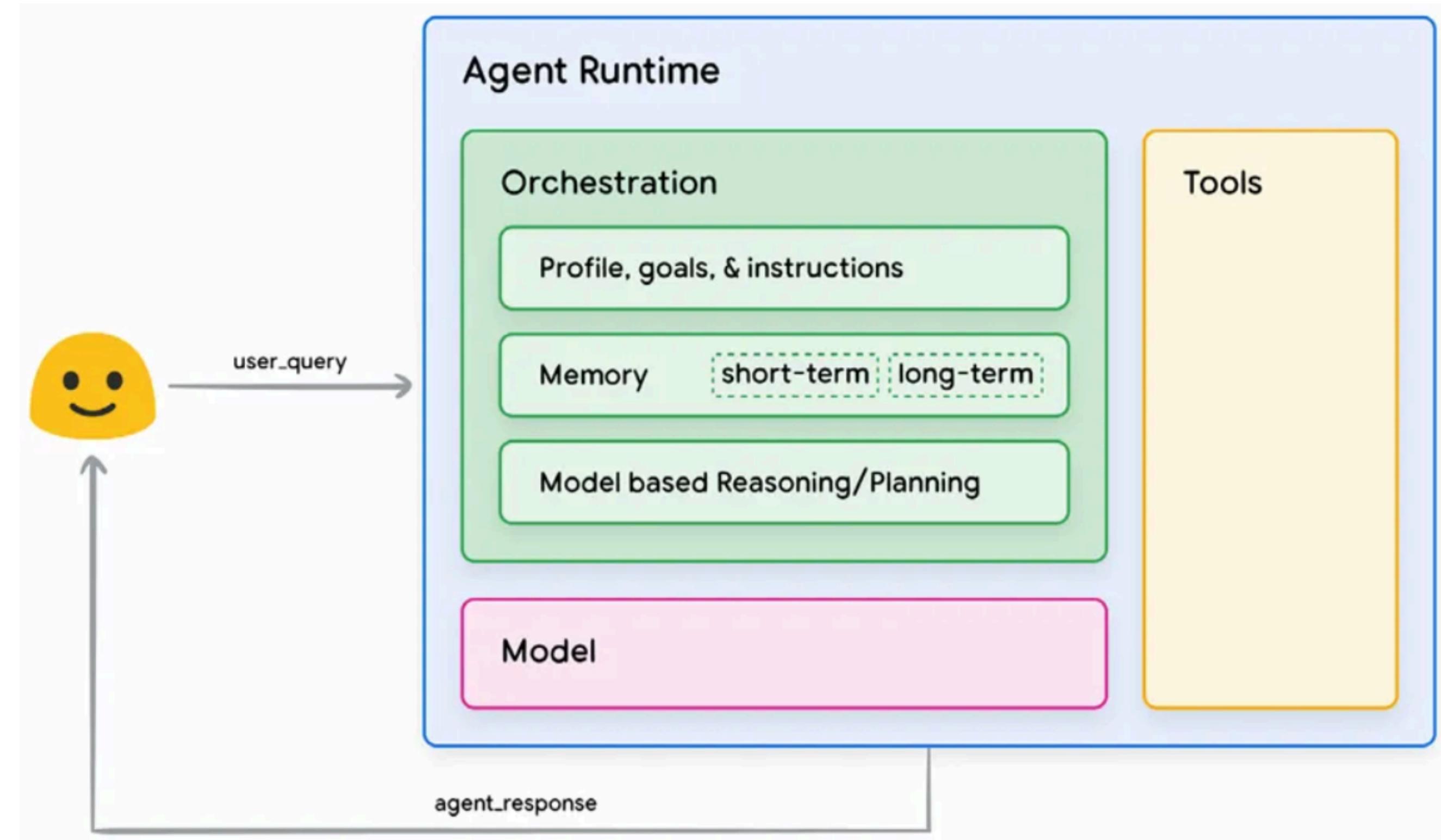

What is Agent



CONTENTS

- 1 소개
- 2 도구
- 3 모델 성능 향상
- 4 Quick start
- 5 Vertex AI agent

01 소개



에이전트는 목표를 달성하기 위해 세상을 관찰하고 자신이 가진 도구를 사용하여 행동하려는 애플리케이션

01 소개

모델



프로세스의 중앙 결정자로 사용될 LM

Tools



외부와 상호작용할 수 없다는 간극을 메움

Orchestration layer



순환 프로세스

01

소개

- **Agent ↔ Model**
 - **지식**: 학습 데이터로 제한 ↔ 도구를 통해 외부 시스템과의 연결로 확장
 - **세션**: 단일 추론은 사용자 쿼리에 기반하며 세션 기록이나 연속적인 컨텍스트 없음 ↔ 사용자 쿼리나 Orchestration layer에서 내린 결정에 기반한 멀티턴 추론이 허용되는 세션 기록 존재
 - **추론**: reasoning framework를 사용해야만 복잡한 논리 layer 사용 ↔ 원시 cognitive architecture에 포함되어 있음

01

소개

Cognitive Architecture

에이전트는 정보를 반복적으로 처리하고, 정보에 입각한 결정을 내리고, 이전 출력을 바탕으로 다음 행동을 정제함으로써 cognitive architecture를 사용하여 최종 목표에 도달한다.

메모리, 상태, 추론 및 계획을 유지하는 오케스트레이션 레이어가 존재한다.

ReAct

Chain-of-Thought (CoT)

Tree-of-thoughts (ToT)

당신은 문제를 해결할 때 다음 형식을 따르세요 :

Thought: ...

Action: <도구이름>[인수...]

Observation: <도구결과>

... (필요 시 반복)

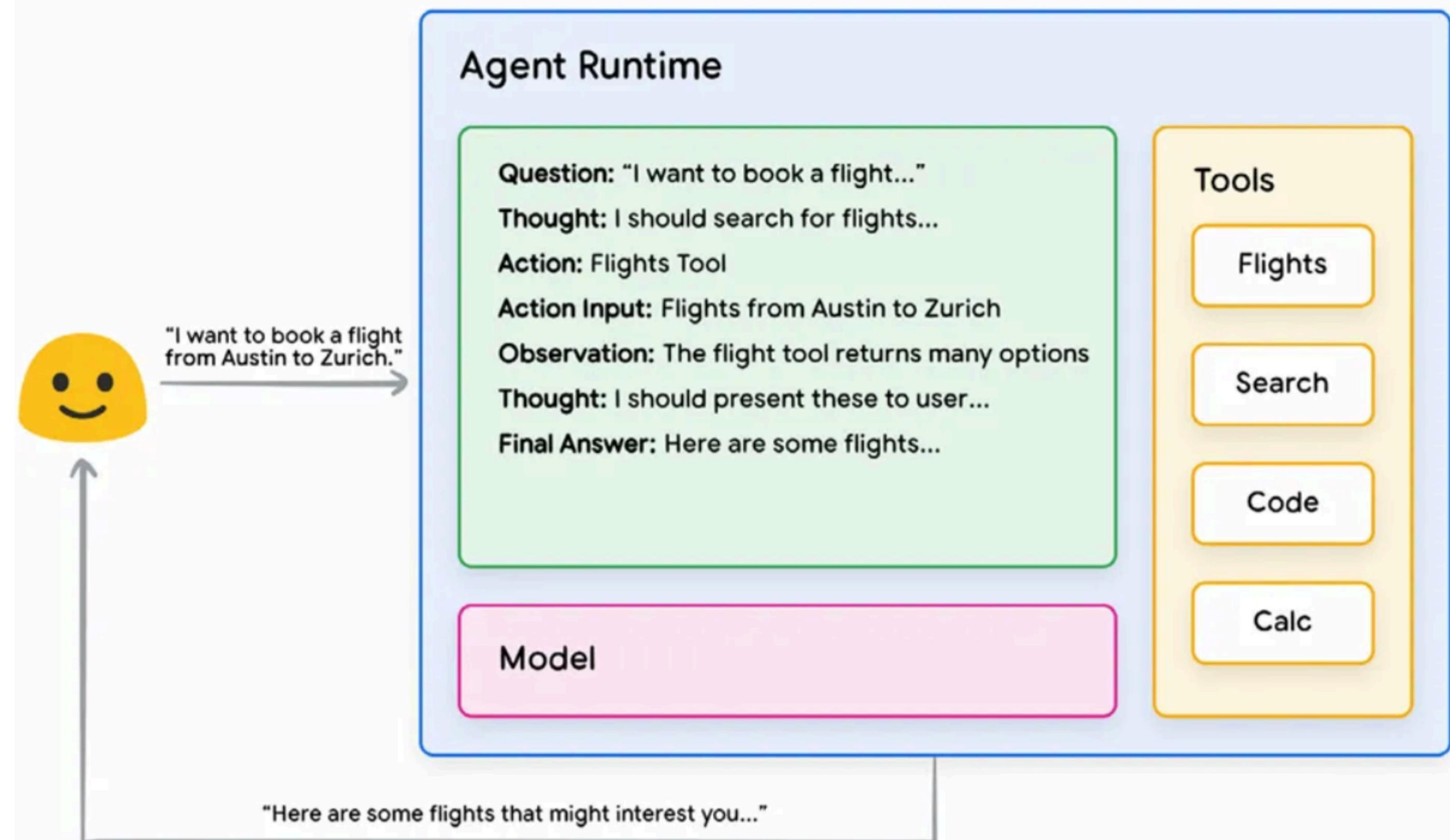
Final Answer: ...

CoT: 프롬프트로 단계별 생각을 쓰게 하고, 필요 시 self-consistency + 검증기로 고른다.

ToT: 분기 생성→평가→탐색 루프를 코딩해 모델을 트리 탐색에 참여시킨다.

01

소개



02 도구

Extensions



02

도구 Extensions

```
import vertexai
import pprint

PROJECT_ID = "YOUR_PROJECT_ID"
REGION = "us-central1"

# Vertex AI 초기화
vertexai.init(project=PROJECT_ID, location=REGION)

from vertexai.preview.extensions import Extension

# 코드 인터프리터 확장 불러오기
extension_code_interpreter = Extension.from_hub("code_interpreter")

# 요청할 코드 내용
CODE_QUERY = """이진 트리를 O(n) 시간 복잡도로 뒤집는 파이썬 메서드를 작성하세요."""

# 확장 실행
response = extension_code_interpreter.execute(
    operation_id="generate_and_execute",
    operation_params={"query": CODE_QUERY}
)

print("생성된 코드:")
pprint.pprint(response['generated_code'])
```

```
# 위의 코드가 아래 코드(예시)를 자동으로 생성
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def invert_binary_tree(root):
    """
    이진 트리를 뒤집는 함수

    매개변수:
        root: 이진 트리의 루트 노드
    반환값:
        뒤집힌 이진 트리의 루트 노드
    """
    if not root:
        return None

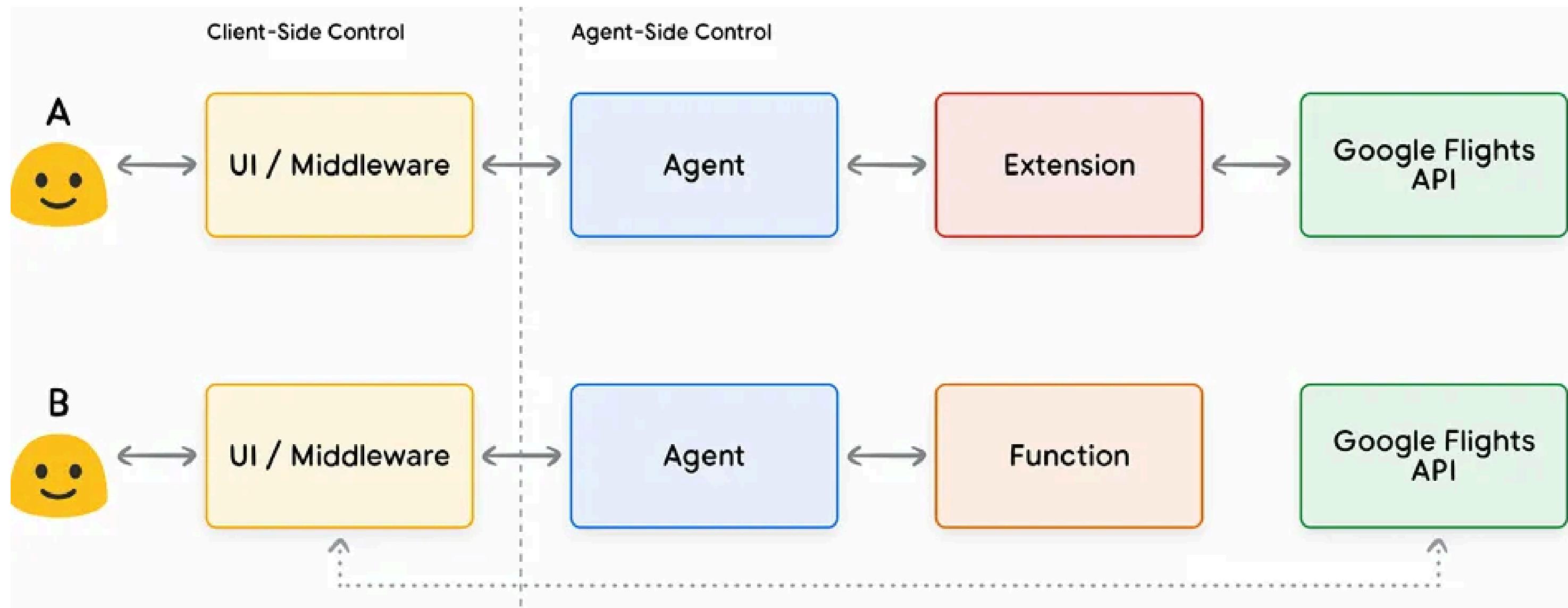
    # 왼쪽과 오른쪽 자식을 재귀적으로 바꿔줌
    root.left, root.right = invert_binary_tree(root.right), invert_binary_tree(root.left)

    return root

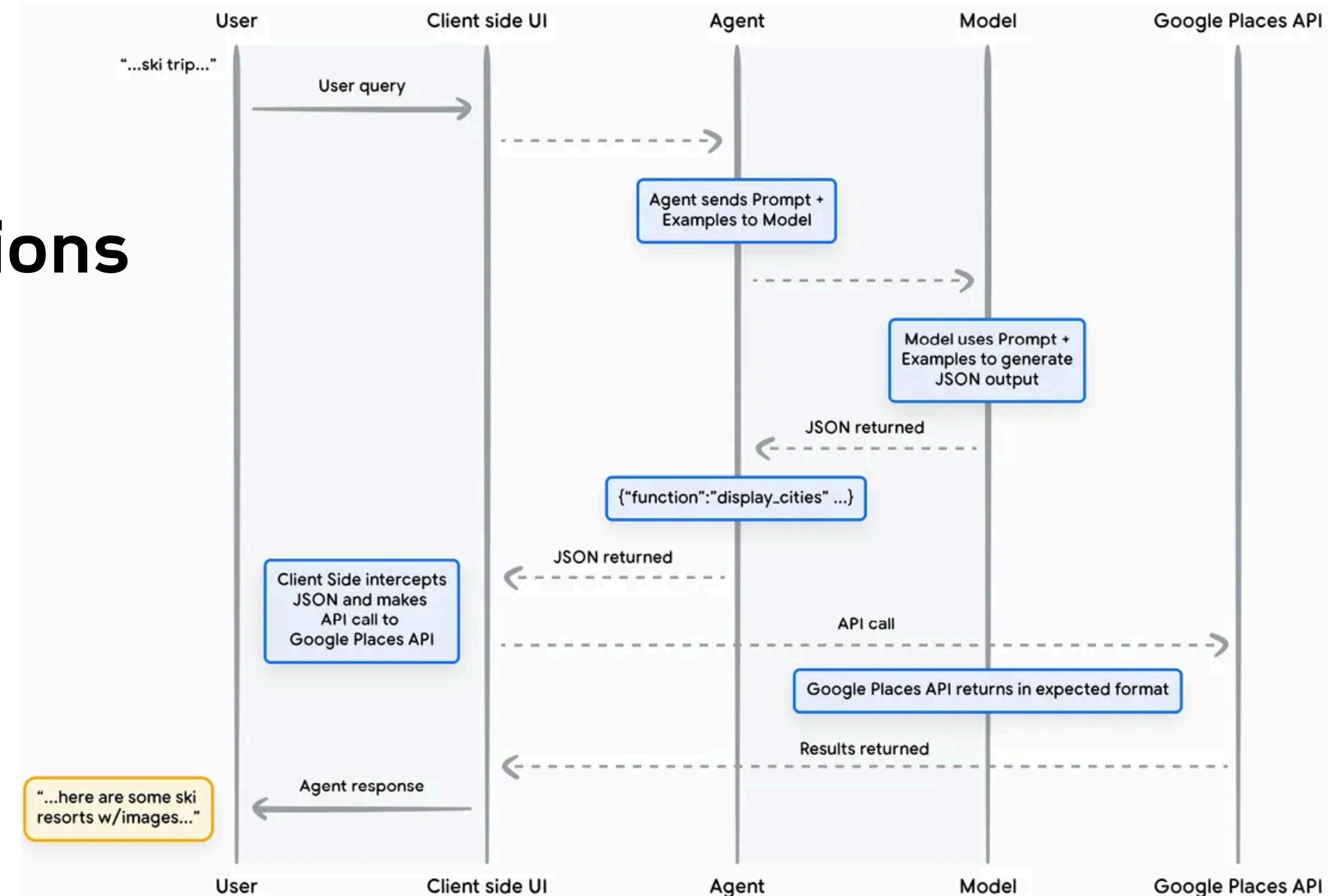
# 사용 예시:
# 샘플 이진 트리 생성
root = TreeNode(4)
root.left = TreeNode(2)
root.right = TreeNode(7)
root.left.left = TreeNode(1)
root.left.right = TreeNode(3)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)

# 이진 트리 뒤집기
inverted_root = invert_binary_tree(root)
```

02 도구 Functions



02 도구 Functions



02 도구

Functions

```
from typing import Optional

# 사용자의 검색어 및 선호도에 따라 도시 목록을 제공하는 함수
def display_cities(cities: list[str], preferences: Optional[str] = None):
    """
    사용자의 검색 조건과 선호도에 따라 도시 목록을 제공합니다.

    매개변수:
        preferences (str): 사용자의 검색 선호도 (예: 스키, 해변, 맛집, BBQ 등)
        cities (list[str]): 추천할 도시 목록

    반환값:
        list[str]: 추천된 도시 목록
    """

    return cities

from google.genai import Client, types

# Vertex AI를 사용하는 클라이언트 설정
client = Client(
    vertexai=True,
    project="PROJECT_ID",
    location="us-central1"
)
```

```
# Gemini 모델 호출
res = client.models.generate_content(
    model="gemini-2.0-flash-001",
    model_input="가족과 함께 스키 여행을 가고 싶은데 어디로 가면 좋을까요?",
    config=types.GenerateContentConfig(
        tools=[display_cities], # display_cities 함수를 도구로 등록
        automatic_function_calling=types.AutomaticFunctionCallingConfig(disable=True),
        tool_config=types.ToolConfig(
            function_calling_config=types.FunctionCallingConfig(mode='ANY')
        )
    )
)

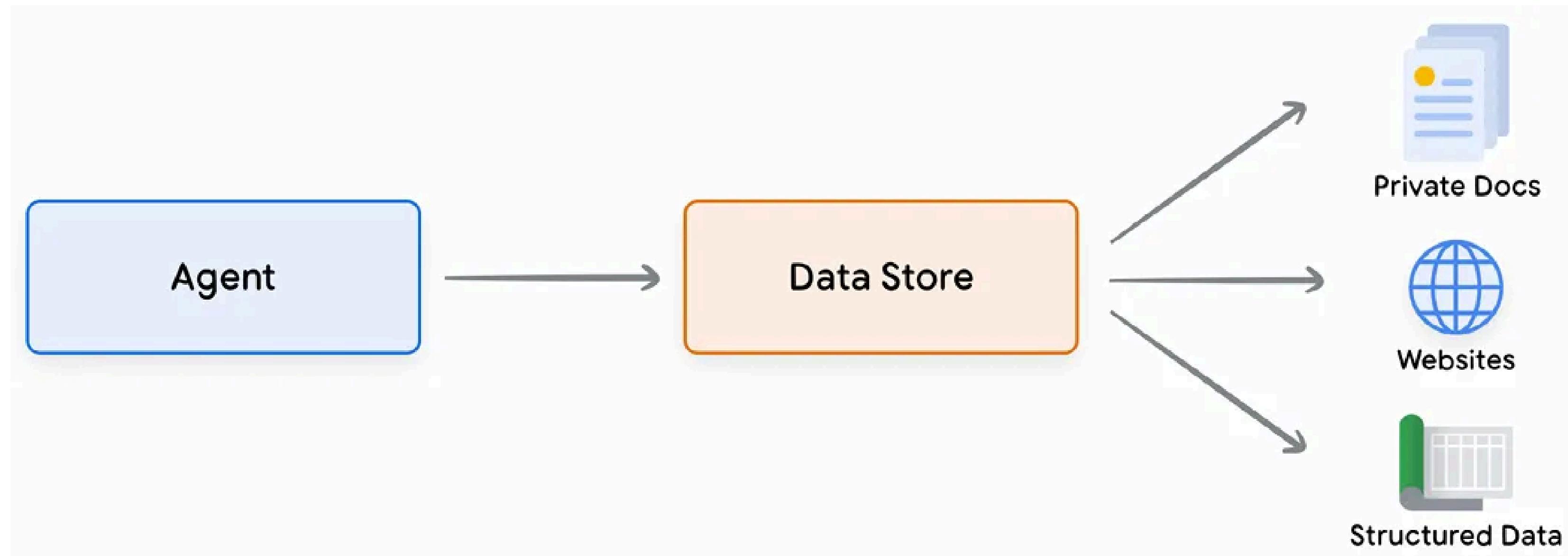
# 모델이 호출한 함수 이름과 전달된 인자를 출력
print(f"함수 이름: {res.candidates[0].content.parts[0].function_call.name}")
print(f"함수 인자: {res.candidates[0].content.parts[0].function_call.args}")

# 출력 예시:
# 함수 이름: display_cities
# 함수 인자: {'preferences': 'skiing', 'cities': ['Aspen', 'Park City', 'Whistler']}
```

02

도구

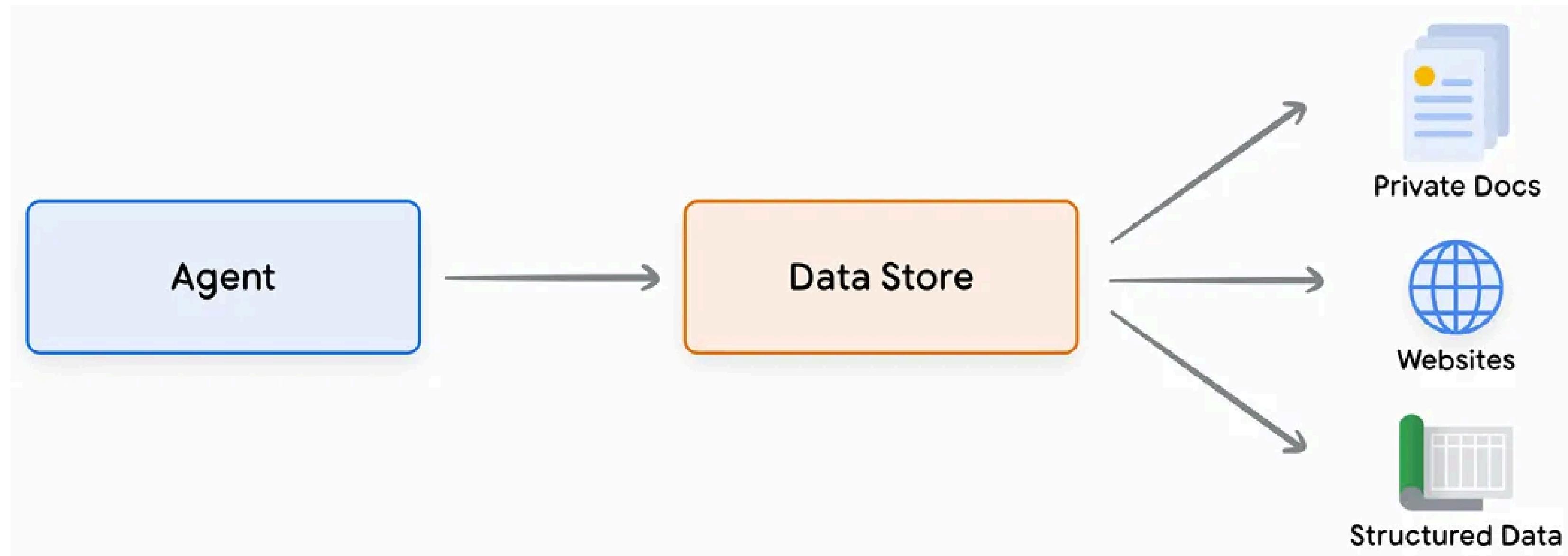
Data stores



02

도구

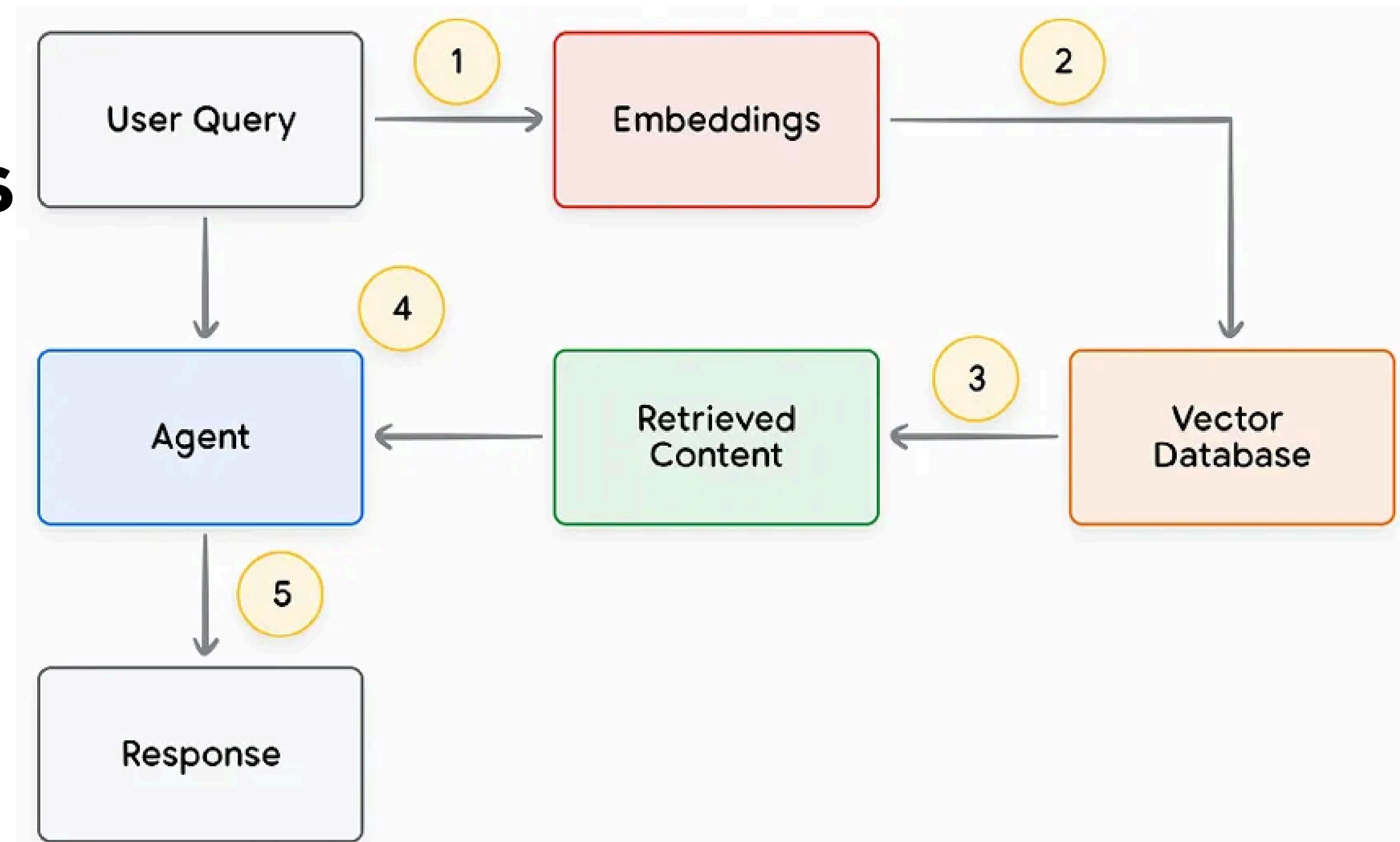
Data stores



02

도구

Data stores



03

모델 성능 향상

In-context learning

추론 시 프롬프트, 도구 및 few shot 예제를 제공하여 일반화된 모델을 통해 '즉석에서' 특정 작업에 대해 어떻게 및 언제 도구를 사용할지 학습한다.

Retrieval-based in-context learning

외부 메모리에서 검색하여 모델 프롬프트에 가장 관련성 높은 정보, 도구 및 관련 예제를 동적으로 채워준다.

Fine-tuning based learning

추론 전 특정 예제의 큰 데이터셋을 사용하여 모델을 학습한다. 이는 모델이 사용자 쿼리를 받기 전에 어떤 도구를 언제 어떻게 적용해야 하는지 이해하도록 돋는다.

04

Quick start

```
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import tool
from langchain_community.utilities import SerpAPIWrapper
from langchain_community.tools import GooglePlacesTool

import os

# 환경 변수(API 키) 설정
os.environ["SERPAPI_API_KEY"] = "XXXXXX"
os.environ["GPLACES_API_KEY"] = "XXXXXX"

# SerpAPI를 이용한 구글 검색 함수
@tool
def search(query: str):
    """SerpAPI를 사용하여 Google 검색을 수행합니다."""
    search = SerpAPIWrapper()
    return search.run(query)

# Google Places API를 이용한 장소 검색 함수
@tool
def places(query: str):
    """Google Places API를 사용하여 장소 정보를 조회합니다."""
    places = GooglePlacesTool()
    return places.run(query)
```

```
# Gemini 모델 설정
model = ChatVertexAI(model="gemini-2.0-flash-001")

# 사용할 도구(tool) 목록
tools = [search, places]

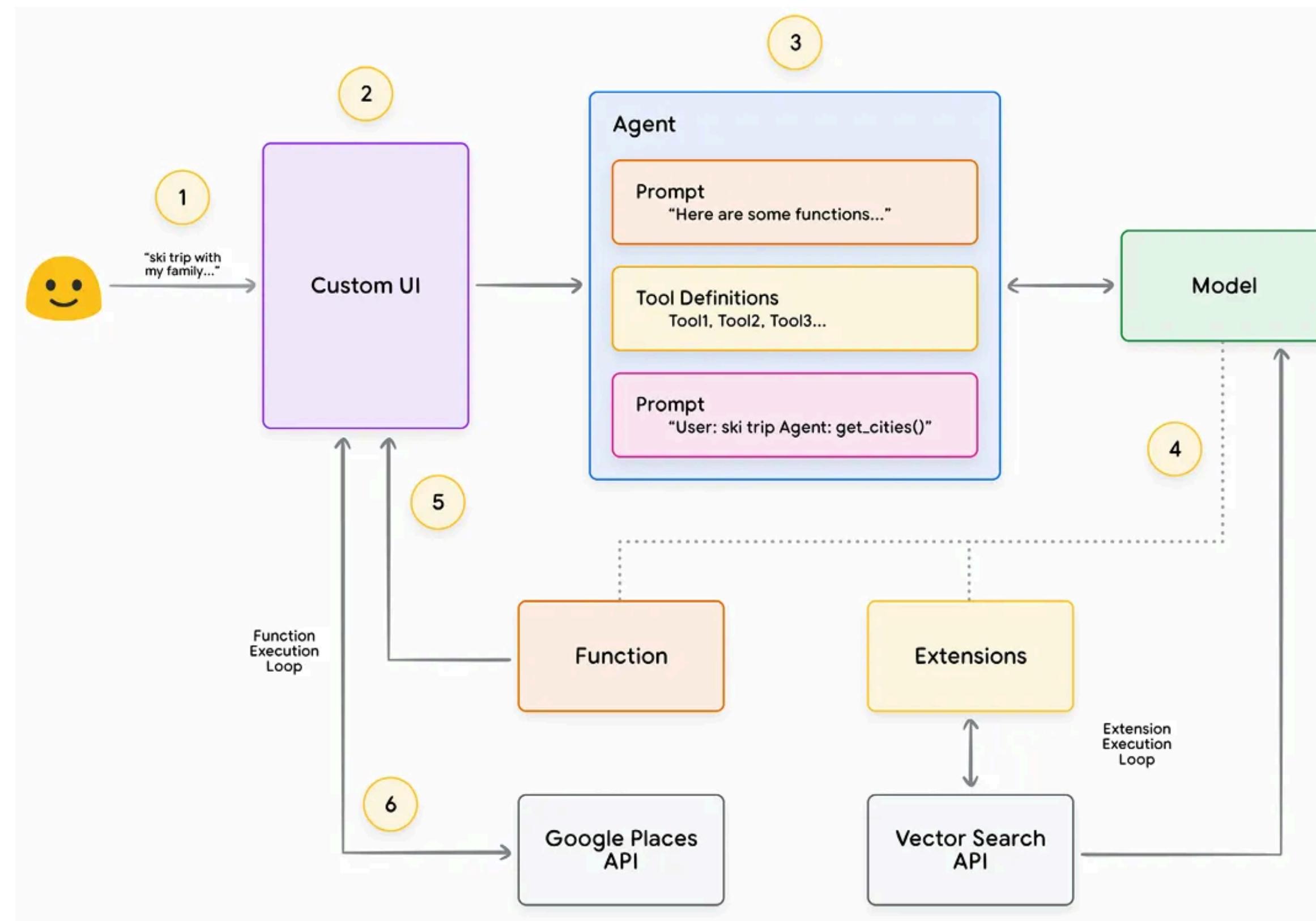
# 모델에 전달할 질문 (자연어 입력)
query = "지난주에 텍사스 롱호너스(Texas Longhorns)가 뜬 경기를 한 상대 팀은 누구인가요? 그 팀의 경기장 주소는 무엇인가요?"

# LLM(대형 언어 모델)과 도구를 연결하는 리액트 에이전트 생성
agent = create_react_agent(model, tools)

# 사용자 입력 구성
input = {"messages": [("human", query)]}

# 에이전트가 응답을 스트리밍 형태로 출력하도록 설정
for s in agent.stream(input, stream_mode="values"):
    message = s["messages"][-1]
    if isinstance(message, tuple):
        print(message)
    else:
        message.pretty_print()
=====
Human Message =====
Who did the Texas Longhorns play in football last week? What is the address of the other team's stadium?
=====
Ai Message =====
Tool Calls: search
Args:
    query: Texas Longhorns football schedule
=====
Tool Message =====
Name: search
{...Results: "NCAA Division I Football, Georgia, Date..."}
=====
Ai Message =====
The Texas Longhorns played the Georgia Bulldogs last week.
Tool Calls: places
Args:
    query: Georgia Bulldogs stadium
=====
Tool Message =====
Name: places
{...Sanford Stadium Address: 100 Sanford...}
=====
Ai Message =====
The address of the Georgia Bulldogs stadium is 100 Sanford Dr, Athens, GA 30602, USA
```

05 Production application



00 정리

1. **에이전트는 도구를 활용하여 실시간 정보에 접근하고 실제 세계의 행동을 제안하며 복잡한 작업을 자율적으로 계획하고 실행하여 언어 모델의 능력을 확장한다.** 에이전트는 하나 이상의 언어 모델을 활용하여 상태를 전환할 때와 어떻게 전환할지를 결정하고, 모델이 스스로 완료하기 어렵거나 불가능한 복잡한 작업을 수행하기 위해 외부 도구를 활용할 수 있다.
2. **에이전트의 작동 핵심은 오케스트레이션 레이어로,** 이는 추론, 계획, 의사 결정을 구조화하고 그 행동을 안내하는 cognitive architecture이다. ReAct, Chain-of-Thought, Tree-of-Thoughts와 같은 다양한 추론기법은 오케스트레이션 레이어가 정보를 입력받고 내부 추론을 수행하여 정보에 입각한 의사결정이나 응답을 생성할 수 있는 프레임워크를 제공한다.
3. Extension, Function, Data store와 같은 도구는 **에이전트가 외부 시스템과 상호 작용하고 훈련 데이터를 넘어선 지식에 접근하는데 외부 세계의 열쇠로서 기능한다.** Extension은 에이전트와 외부 API간의 다리를 제공하여 API 호출의 실행과 실시간 정보의 검색을 가능하게 한다. Function는 개발자를 위한 더 상세한 제어를 제공하여 에이전트가 함수 매개변수를 생성하고 클라이언트측에서 실행할 수 있게 한다. Data store는 에이전트가 구조화되거나 비구조화된 데이터에 접근할 수 있게 하여 데이터 기반 애플리케이션을 가능하게 한다.

THANK YOU
