Installation der Pakete

```
# Offizielles OpenAI-SDK, um über Python mit den OpenAI-APIs zu kommunizieren
!pip install openai

# HTTP-Client-Bibliothek für Python, um Web-APIs wie OpenWeatherMap abzufragen
!pip install requests

# Bibliothek für Datenvalidierung/-
modellierung, die dafür sorgt dafür, dass Ausgaben in einem festen Schema (Structured
Output) vorliegen
!pip install pydantic

# Hilfsbibliothek, um Listen/Tabellen als schön formatierte Texttabellen auszugeben
!pip install tabulate
```

Imports und Basis-Setup

```
# Openai-client-klasse importieren (für Chat-Completion Aufrufe)
from openai import OpenAI

# Pydantic-Basisklasse für strikt typisierte/validierte Datamodelle
from pydantic import BaseModel

# Sorgt für strukturierte Tabellen in der Konsole
from tabulate import tabulate

# Datum und Zeit für Zeitstempel
from datetime import datetime

# Lädt 'requests' für HTTP-Anfragen (z.B. API-
Aufrufe) und 'json' zum Umwandeln zwischen JSON-Text und Python-Datenstrukturen
import requests, json

# OpenAI-Key laden
from google.colab import userdata
OPENAI_API_KEY = userdata.get('apikey_ab')

# OpenWeatherMap API-Key
OWM_API_KEY = "05c40e2e21ab66e6e5ebf30526e49c3a"

# Standard-Stadt für die Abfrage
CITY = "Stuttgart"

# OpenAI-Client initialisieren
client = OpenAI(api_key=OPENAI_API_KEY)
```

Tool-Nutzung: Wetter-Tool

```
# Definiert eine Funktion, die für eine Stadt aktuelles Wetter von OpenWeatherMap abruft
def get_weather(city: str):
    # Sendet eine HTTP-GET-Anfrage an die OWM-
API mit Stadtname, metrischen Einheiten, deutscher Sprache und API-Key
    resp = requests.get(
        f"http://api.openweathermap.org/data/2.5/weather?q={city}
&units=metric&lang=de&appid={OWM_API_KEY}",
        timeout=15 # Beendet den Request automatisch, wenn nach 15 Sekunden keine
```

```
Antwort kommt
    )

    # Wandelt die API-Antwort aus JSON in ein Python-Objekt (Dictionary) um
    data = resp.json()
    # Prüft, ob die Antwort fehlerfrei ist (HTTP-Code & API-Code)
    if resp.status_code != 200 or (data.get("cod") not in (200, None)):
        msg = data.get("message", f"HTTP {resp.status_code}") # Holt die Fehlermeldung aus
der Antwort oder nutzt den HTTP-Statuscode
        raise RuntimeError(f"Fehler beim Abruf: {msg}") # Bricht die Funktion mit einer
Fehlermeldung ab, wenn die Abfrage scheitert

    # Liest die aktuelle Temperatur in °C aus den Wetterdaten aus
    temp = float(data["main"]["temp"])

    # Liest die Kurzbeschreibung des aktuellen Wetters aus
    desc = data["weather"][0]["description"]

    # Erstellt ein Dictionary mit heutigem Datum und Temperatur
    point = {"date": datetime.now().strftime("%Y-%m-%d"), "temp_c": temp}

    # Gibt die Wetterdaten als strukturiertes Dictionary zurück
    return {"city": city, "temperature_c": temp, "description": desc, "points": [point]}


# Tool-Spezifikation für das LLM (Function Calling)
tools = [{ # Definiert eine Liste mit einem Tool, das das LLM aufrufen darf
    "type": "function", # Legt fest, dass es sich bei diesem Tool um eine Funktion handelt
    "function": {
        "name": "get_weather", # Name des Tools, der im LLM-Aufruf referenziert wird
        "description": "Get current weather for a city (temperature + short description).", #
Beschreibung für das LLM, wofür das Tool gedacht ist
        "parameters": {  # Legt das JSON-
Schema für die Parameter fest, die das LLM übergeben muss
            "type": "object", # Parameter werden als JSON-Objekt übergeben
            "properties": {"city": {"type": "string"}}, # Erlaubtes Feld: "city" als String
            "required": ["city"], # Das Feld "city" ist Pflicht
            "additionalProperties": False # Keine weiteren Felder außer den definierten sind
erlaubt
        },
        "strict": True # Erzwingt strikte Einhaltung des Schemas beim Funktionsaufruf
    }
}]
```

Prompt-Chaining

```
# 1. Schritt: Erster LLM-Call entscheidet, ob/wie das Tool aufzurufen ist

# Erstellt die Nachrichtenliste, die als Gesprächskontext an das LLM geschickt wird
messages = [
    # Systemnachricht: definiert Rolle und Stil der Antworten
    {"role": "system", "content": "You are a concise weather assistant. Keep outputs short."},
    # Benutzerfrage mit der gewünschten Stadt (CITY)
    {"role": "user", "content": f"What's the current weather in {CITY}?"}
]

# Ruft das OpenAI-API auf, um eine Chat-Antwort vom Modell zu generieren
```

```
c1 = client.chat.completions.create(
    model="gpt-4o-mini", # Modellauswahl
    messages=messages, # Übergibt den Gesprächskontext an das Modell
    tools=tools, # Übergibt die Liste verfügbarer Tools, die das Modell aufrufen darf
    max_tokens=200 # Beschränkt die maximale Länge der Modellantwort
)

# 2. Schritt: Tool tatsächlich ausführen und Ergebnis zurück in den Chat geben

# Iteriert über alle vom Modell vorgeschlagenen Tool-Aufrufe
for tc in c1.choices[0].message.tool_calls or []:
    messages.append(c1.choices[0].message) # Fügt die Assistant-Nachricht mit dem Tool-
Aufruf zum Nachrichtenverlauf hinzu
    args = json.loads(tc.function.arguments) # Parst die vom Modell als JSON gesendeten
Argumente in ein Python-Dictionary
    result = get_weather(**args)  # Führt das Wetter-
Tool mit den Argumenten aus und speichert das Ergebnis
    messages.append({"role": "tool", "tool_call_id": tc.id, "content": json.dumps(result)})  #
Fügt das Tool-Ergebnis als 'tool'-Nachricht in den Nachrichtenverlauf ein
```

Structured Output

```
# Definiert ein Pydantic-Datenmodell für einen einzelnen Wetterdatenpunkt
class WeatherPoint(BaseModel):
    date: str                # Datum als Zeichenkette
    temp_c: float            # Temperatur in Grad Celsius als Fließkommazahl

# Definiert ein Pydantic-Datenmodell für den kompletten Wetterbericht
class WeatherReport(BaseModel):
    city: str                    # Name der abgefragten Stadt
    temperature_c: float         # Aktuelle Temperatur in Grad Celsius
    description: str             # Kurze Wetterbeschreibung
    points: list[WeatherPoint] = []  # Liste mit Wetterpunkten (hier nur ein Eintrag für heute)
    avg_temp_c: float            # Durchschnittstemperatur (kann vom Modell berechnet
werden)
    note: str                    # Kurze Empfehlung oder Bemerkung basierend auf der
Temperatur

# 3. Schritt: Zweiter LLM-Call: finale Antwort erzeugen und strikt ins Schema parsen

# Ruft das OpenAI-API auf und parsed die Antwort direkt ins WeatherReport-Schema
c2 = client.beta.chat.completions.parse(
    model="gpt-4o-mini",             # Modellwahl
    messages=messages,               # Gesamter Nachrichtenverlauf (inkl. Tool-
Ergebnis) als Kontext
    tools=tools,                     # Liste der verfügbaren Tools
    response_format=WeatherReport,   # Erwartetes Ausgabeformat: Instanz des
WeatherReport-Pydantic-Modells
    max_tokens=200                   # Maximale Länge der Antwort in Tokens begrenzen
)
final = c2.choices[0].message.parsed      # Extrahiert die geparste Modellantwort als
WeatherReport-Objekt


# Ausgabe bestehend aus JSON und TABELLE

print("JSON Output:")                    # Überschrift für die JSON-Ausgabe
```

```
print(final.model_dump_json(indent=2))    # Gibt den WeatherReport formatiert als JSON-String aus (mit Einrückung)

# Prüft, ob die Liste der Wetterpunkte nicht leer ist
if final.points:
    table = [["Datum", "Temp (°C)"]] + [[p.date, p.temp_c] for p in final.points]  # Baut eine Tabellenstruktur aus den Wetterpunkten
    print("\nTabelle:")                                          # Überschrift für die Tabelle
    print(tabulate(table, headers="firstrow", tablefmt="github"))                # Gibt die Tabelle im Markdown-kompatiblen Format aus
```